

RL-TR-93-113
Final Technical Report
June 1993

AD-A269 191



2
108

SOFTWARE ENGINEERING FOR EFFECTIVE UTILIZATION OF PARALLEL PROCESSING COMPUTING SYSTEM

University of Florida

Stephen S. Yau, Doo-Hwan Bae, Madhan Chidambaram,
Gilda Pour, Venkeepuram R. Satish, Wai-Kong Sung,
and Keunhyuk Yeom

DTIC
ELECTE
SEP 13 1993
S B D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

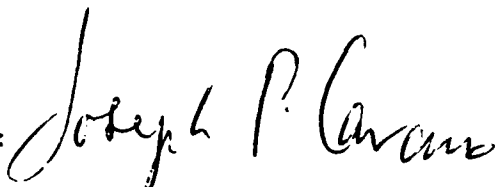
93-21224 140pf
93-21224 140pf
93-21224 140pf

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

93 9 13 02 0

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations. RL-TR-93-113 has been reviewed and is approved for publication.

APPROVED:



JOSEPH P. CAVANO
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO
Chief Scientist
Command, Control and Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE June 1993		3. REPORT TYPE AND DATES COVERED Final May 91 - Sep 92	
4. TITLE AND SUBTITLE SOFTWARE ENGINEERING FOR EFFECTIVE UTILIZATION OF PARALLEL PROCESSING COMPUTING SYSTEM				5. FUNDING NUMBERS C - F30602-91-C-0045 PE - 63728F PR - 2527 TA - 03 WU - P7	
6. AUTHOR(S) Stephen S. Yau, Doo-Hwan Bae, Madhan Chidambaram, Gilda Pour, Venkeepuram R. Satish, Wai-Kong Sung and Keumhyuk Yeom					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Florida Computer and Information Sciences Department Gainesville FL 32611				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3CB 525 Brooks Road Griffiss AFB NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-93-113	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Joseph P. Cavano/C3CB/(315)330-4063.					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This effort produced an approach for developing software for parallel processing systems based on the parallel object-oriented functional computational model (PROOF). A benefit of this approach is that it allows the software development to be separated from architecture-dependent issues to a great degree. For example, the programmers do not need to be concerned with issues such as synchronization, parallelization, or the topology of the parallel processing system when they address objects in the system. The object-oriented paradigm reflects the parallel structure of the problem space and can represent inherently concurrent behavior. This approach is suitable for MIMD machines and should result in software that is more understandable, modifiable and portable over a variety of parallel computer architectures.					
14. SUBJECT TERMS Parallel Software Engineering, Parallel Processing, Parallel Software Development				15. NUMBER OF PAGES 144	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT U/L		

Abstract

An approach is developed for software development for parallel processing systems based on the parallel object-oriented functional computational model (PROOF) which incorporates the functional paradigm into the object-oriented paradigm. Our approach separates the architecture-dependent issues from the software development, and consequently enhances the portability of the software. This approach facilitates the software development for any parallel processing system by freeing the programmer from the consideration of various parallelization aspects of the software, and the network topology of the parallel processing system. It allows the exploitation of parallelism at various levels of granularity: object level and method level thereby making it suitable for the development of software for any MIMD machines. Furthermore, this approach retains the benefits of both the object-oriented and the functional paradigms, such as modifiability, understandability, portability, and parallelizability. A framework for software development for parallel processing systems which consists of object-oriented analysis, object design, coding and transformation phases has been established. Software development is done in a high-level prototype parallel language PROOF/L based on the computation model PROOF and then the code in PROOF/L is transformed to a target language of the MIMD machine. The transformation of the code in PROOF/L to a target language is performed via a two-step translation: one from the code written in PROOF/L to an intermediate form, and the other from the intermediate form to the code in the target language. An example is given to illustrate this approach.

Keywords- Parallel processing systems, MIMD machine , software development framework, object-oriented analysis, object design, verification, partitioning, grain size determination, PROOF, PROOF/L, transputers.

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Contents

1	Introduction	1
2	Background	3
2.1	Overview of Parallel Programming Approaches	3
2.2	PROOF and PROOF/L	4
3	Overall Framework	9
3.1	Object-oriented Analysis	9
3.1.1	Identifying Objects and Classes	10
3.1.2	Determining Class Interfaces	11
3.1.3	Specifying Dependency and Communication Relationships Among Objects	11
3.1.4	Identifying Active, Passive and Pseudo-Active Objects	12
3.1.5	Identifying Shared Objects	12
3.1.6	Specifying the Behavior of Objects	13
3.1.7	Identifying Bottleneck Objects	14
3.1.8	Checking For Completeness and Consistency	14
3.2	Object Design	15
3.2.1	Establishing Class Hierarchy	15
3.2.2	Designing Class Composition and Methods	15
3.2.3	Designing the Bodies for Active and Pseudo-Active Objects	17
3.3	Verification	18

3.3.1	Transformation of bodies to Petri nets	19
3.3.2	Composition of the Petri-nets	19
3.3.3	Refinement of Petri nets	20
3.4	Coding in PROOF/L	21
3.5	Transformation from PROOF/L to any Target Language	23
4	Partitioning	24
4.1	Our Partitioning Approach	25
4.1.1	Initialization	25
4.1.2	Normalization	28
4.1.3	Clustering	29
4.2	Time Complexity	31
5	PROOF/L Front-end Translation	32
5.1	Syntax Rules of PROOF/L	32
5.2	Intermediate Program Representation	35
5.3	Translation Rules : From PROOF/L to IPR	36
5.4	Textual Form of IPR	41
5.5	Implementation of PROOF/L Front-End Translator	44
5.5.1	Lexical Analyzer	45
5.5.2	Parser	46
5.5.3	Code Generation	47
5.5.4	Symbol Table Handling	47
5.6	Limitations	48
6	Grain Size Analysis	50
6.1	Grain Size Determination	51
6.2	Tree Parallelism	53
6.3	Graph Parallelism	62

6.4	Pipelined Parallelism	64
6.5	Modifying Intermediate Form	70
6.5.1	Modifying Intermediate Form Using Partitioning Information . .	71
6.5.2	Modifying Intermediate Form Using Grain Size Analysis	72
7	PROOF/L Back-end Translation	74
7.1	Translation Rules	74
7.1.1	Simple Structure	74
7.1.2	Schemes for Detecting Patterns	80
7.2	An Algorithm Traversing the Data Dependency Graph	81
7.3	Examples	83
7.4	Implementation of the Back-end Translator	86
7.4.1	Reading the Input	87
7.4.2	Internal Representation of Data Dependency	89
7.4.3	Code Generation of Method and Objects	90
7.4.4	Translation of Nodes	92
8	An Application Example	93
8.1	Specifications for the Defense of Air Force Bases	93
8.2	Object-Oriented Analysis	95
8.2.1	Identifying Classes and Objects	95
8.2.2	Defining Class Interfaces	95
8.2.3	Specifying Dependency and Communication Relationships Among Objects	96
8.2.4	Identifying Active, Passive and Pseudo-Active Objects	98
8.2.5	Identifying Shared Objects	98
8.2.6	Specifying the Behavior of Each Object	98
8.2.7	Identifying Bottleneck Objects	100

8.2.8	Checking for Completeness and Consistency of the Object-Oriented Analysis	102
8.3	Object Design	103
8.3.1	Establishing Hierarchy	103
8.3.2	Designing Class Composition and Methods	104
8.3.3	Designing the Body of the Objects	106
8.4	Verification	106
8.5	Partitioning	107
8.6	Implementation	120
8.6.1	Coding	120
8.6.2	Grain Size Analysis	120
8.6.3	Interfacing With the Router	121
9	Conclusions and Future Research	122
	Bibliography	124

List of Figures

3.1	The various phases of our framework.	10
3.2	An object communication diagram.	12
3.3	Transformation of the control constructs.	20
3.4	Transformation of a method requiring modification.	20
3.5	Combining two objects with a common bottleneck place.	21
3.6	Refinement of the methods with the guards.	22
3.7	Refinement of the methods after coding.	22
5.1	The IPR forms of select, distributor and merge node.	36
5.2	The corresponding IPR for <i>function application</i>	36
5.3	The corresponding IPR for <i>list</i>	37
5.4	The corresponding IPR for α functional form.	37
5.5	The corresponding IPR for β functional form.	37
5.6	The corresponding IPR for γ functional form.	38
5.7	The corresponding IPR for <i>lambda</i> abstraction.	38
5.8	The corresponding IPR for <i>while</i> functional form.	38
5.9	The corresponding IPR for <i>if</i> functional form.	39
5.10	The corresponding IPR for <i>spilt</i> node insertion.	39
5.11	Functions f_1, f_2, \dots, f_n have the same input x	40
5.12	The resulting IPR after a <i>copy</i> node is inserted.	40
5.13	The corresponding IPR for <i>sequential</i> functional form.	40

5.14	The input/output of a <i>latch</i> node.	41
5.15	A typical IPR node.	41
5.16	IPR for method <i>get</i>	43
5.17	IPR for method <i>put</i>	44
5.18	Architecture of the PROOF/L Front-End Translator.	44
5.19	The structure of the symbol table.	48
6.1	An IPR representation for Φ_1	52
6.2	Two simple gain tree examples: (a) in-tree form, and (b) out-tree form	56
6.3	The task precedence tree of the tree parallelism example.	58
6.4	The schedule obtained from McCreary approach.	58
6.5	A gain tree for the tree parallelism example.	59
6.6	A schedule obtained from our approach: (a) for 4 processors, and (b) for 5 processors.	60
6.7	A task precedence graph for the Fast Fourier Transformation problem.	65
6.8	A gain graph for the Fast Fourier Transformation problem.	66
6.9	A schedule for the FFT problem.	67
6.10	An example illustrating the output of partitioning.	71
6.11	A modified intermediate form graph based on partitioning.	72
6.12	A modified intermediate form after grain size determination	73
7.1	The control structures for <i>iteration</i> and <i>predicate</i> statements.	80
7.2	The IPR of <i>get</i> and <i>put</i> methods.	83
7.3	The IPR for <i>factorial</i>	85
7.4	The implementation scheme of a translator for PROOF/L to the target code.	86
7.5	The data structure of a data dependency graph.	89
8.1	The object communication diagram for the set of decomposed objects of the example.	97

8.2	The object communication diagram for the modified set of objects. . .	101
8.3	Transformation of <i>r1</i> to Petri net	107
8.4	Transformation of <i>r2</i> to Petri net	108
8.5	Transformation of <i>r3</i> to Petri net	109
8.6	Transformation of <i>reporter</i> to Petri net	110
8.7	Transformation of <i>b1</i> to Petri net	111
8.8	Transformation of <i>f2</i> to Petri net	112
8.9	Transformation of <i>f2</i> to Petri net (cont.)	113
8.10	Transformation of <i>f3</i> to Petri net	114
8.11	Initial Graph.	117
8.12	Output Graph.	121

List of Tables

8.1	Distribution of aircrafts in the base.	94
8.2	Effectiveness values for the friendly and hostile equipments.	94
8.3	Object Classification	98
8.4	Object classification for the new set of objects	100
8.5	Communication and Concurrency Weights for the Initial Graph.	116
8.6	Weights after Normalization.	118
8.7	Gain Weights after the First Clustering.	118
8.8	Gain Weights after the Second Clustering.	118
8.9	Gain Weights after the Third Clustering.	119
8.10	Gain Weights after the Fourth Clustering.	119
8.11	Weights after the Fifth Clustering.	119
8.12	Weights after the Sixth Clustering.	120
8.13	Weights in Output Graph.	120

Chapter 1

Introduction

The speed of computers has greatly increased during the past decade, especially with the recent rapid development of various commercially available parallel processing systems. Although such vast increase in speed should satisfy the need for high performance computing systems for applications such as C3I (command, control, communication, and intelligence) systems, space exploration mission, weather prediction, and telecommunication systems, where there are many interacting components, shared resources, and computationally intensive tasks, the potential of such high performance computing systems cannot be realized without effective software development methods for such systems. Unfortunately such methods are far from being mature due to the additional complexity of concurrency and synchronization. The lack of such methods is a major obstacle for the use of parallel processing systems in various application areas. The goal of this project is to develop an effective software development approach for parallel processing systems.

In this project we have established a framework for the development of software for parallel processing systems based on the parallel object-oriented functional computational model (PROOF) [1, 2] which incorporates the functional paradigm in the object oriented paradigm. The main advantage of this approach is that this methodology separates the architecture dependent issues from the software development. Hence, the programmer does not need to be concerned with issues such as synchronization, parallelization, or the topology of the parallel processing systems thereby making the software development independent of the architecture of the parallel processing system [2]-[4]. The software developed using this approach is easily portable over a variety of parallel computer architectures. This approach will allow the exploitation of parallelism at various levels of granularity without sacrificing the effectiveness of the object-oriented paradigm. Parallelism is exploited at both the object level (coarse grain) and the method level (fine grain), and hence our approach is suitable for MIMD machines [5, 6]. Software developed based on this paradigm will reflect the parallel structure of the problem space which will make the software more understandable and modifiable.

Our approach to software development for parallel processing systems is to use

a parallel programming language. We have designed and implemented a high level prototype parallel programming language PROOF/L based on PROOF to demonstrate the feasibility of our approach. The software developed using PROOF/L is transformed into a suitable target language supported by the parallel processing system. This transformation is done in two steps: The first involves the translation of PROOF/L code to an intermediate program representation (IPR). This step, known as the PROOF/L front-end translation, makes the implicit parallelism in PROOF/L code explicit. The second involves the translation of the IPR into the target language. This step is known as PROOF/L back-end translation.

In this report, we will briefly summarize various existing approaches to software development for parallel processing systems and provide the necessary background information for PROOF in Chapter 2. Our overall approach to software development for such systems will be elucidated in Chapter 3. An approach to partition the software system into a set of clusters so that coarse grain parallelism can be exploited is given in Chapter 4. In Chapters 5 through 7, we will present the translation from PROOF/L code to the target code. The PROOF/L front-end translation will be described in Chapter 5. In Chapter 7 we will describe the translation process from IPR to target language of the parallel processing system. We will also discuss the implementation issues involved in such a translation. The PROOF/L back-end translation requires additional information regarding the grain size to produce the code which can be executed efficiently on the underlying parallel processing system. In Chapter 6, we will present an approach to grain size analysis on various patterns of parallelism. In Chapter 8 we will give an example to illustrate this approach. Finally, the conclusions and future direction of this research will be given in Chapter 9.

Chapter 2

Background

2.1 Overview of Parallel Programming Approaches

Approaches to programming parallel processing systems can be classified in three categories: One category is to write programs using conventional sequential programming languages, such as Fortran [7, 8, 9] or C [7, 10], and then parallelize the programs using parallelizing or vectorizing compilers. Although this type of approaches seems to be attractive since many existing sequential software can be adapted to such a parallel programming environment with minor modifications, it is hardly effective. The reasons for this are that the parallelizing or vectorizing compilers cannot unravel most of the parallelism, can only detect parallelism associated with iterations over common data structures, such as arrays and matrices [11], and require extensive dependency analysis [12]. Furthermore, because sequential languages are extended with compiler directives in order to help the compilers detect parallelism and because these extensions are machine-specific, portability of programs is hampered. The problems in this type of approaches are not the compilers themselves, but are due to the inherent sequential characteristics of imperative programming languages since these languages are designed for sequential execution in sequential processors. Although this type of approaches is very popular in scientific application areas, it is not promising for various applications of parallel processing systems.

The second category of approaches is to use parallel language constructs to explicitly model the parallelism in programs. These parallel language constructs include the parallel statements and *input*, *output* commands in CSP [13], *monitor* and *wait*, *signal* operations in Concurrent Pascal [14], and *task* and *rendezvous* mechanisms in Ada [15]. Although the imperative languages in this category of approaches are extended with some language constructs, the basic model of computation is still sequential. Using the parallel language constructs will not reduce the programmer's responsibility to explicitly express the parallelism and ensure the correct communication and synchronization among parallel units, which are extremely complex tasks. In addition, these parallel language constructs are only suitable to express coarse grain parallelism. Thus, massive and fine parallelism cannot be expressed in this type of approaches.

The third category of approaches is to use parallel programming languages, such as Id Nouveau [16], and SISAL [17], which are functional languages tailored for scientific computation, PARLOG [18], a parallel logic language, and Act 1 [19], an object-based language based on the Actor model. The underlying computation models of these parallel programming languages are fundamentally different from the underlying models of imperative programming languages in that parallelism is mostly implicit and massive parallelism can easily be obtainable. Hence, the programmer using these languages is liberated from the complications caused by parallelism. This type of approaches is considered most promising in parallel programming. However, the existing approaches based on parallel programming languages have some or all of the following disadvantages:

- Software engineering concepts for managing parallelism have not been fully incorporated in these approaches.
- Most of the approaches are targeted for the shared memory processors.
- The concept of shared data has not been introduced into programming parallel processors.
- Coding of correct synchronization and communication using explicit constructs is still the programmer's responsibility.

Our approach falls in the third category because it uses the parallel programming language PROOF/L. However, our approach will overcome the above difficulties.

2.2 PROOF and PROOF/L

Our approach is to use a parallel programming language PROOF/L based on the computation model PROOF which incorporates the functional paradigm into the object-oriented paradigm. In this section, for the sake of completeness, we will summarize the important features of PROOF and PROOF/L which will be used in our approach. For more detailed information, the reader is referred to [1]

An object-oriented programming model naturally reveals existing parallelism in the application problems [20]. Besides the advantages of modifiability, maintainability and reusability, one significant advantage of the object-oriented model over others is that the concept of an object can be used at earlier stages of software development cycles than the implementation stage. It implies that parallel processing aspects such as parallelism and communication among parallel components can be naturally handled at the earlier stage of the software development. Consequently, it is easy for the programmer to handle parallelism and communication among parallel components. However, in the object-oriented model, parallel execution of concurrent objects is the only source of parallelism, and hence the parallelism to be exploited may not be sufficient for exploiting fine-grain parallelism.

On the other hand, functional languages based on the functional paradigm are referentially transparent and race conditions cannot be introduced. As a result, this paradigm has great potential of exploiting implicit parallelism by removing side-effects caused by assignment statements. Functional programming has been one of the main directions in developing new languages that directly address the challenge of parallel programs, i.e. parallelism can be easily detected and exploited. However, due to its history insensitivity, the expressive power of programming is limited and is not suitable for expressing inherently concurrent nature.

PROOF incorporates the functional paradigm in the object-oriented paradigm by supporting object-oriented features, such as object, class and inheritance, and defining methods as purely applicative functions. Thus, PROOF allows the exploitation of massive parallelism without sacrificing the effectiveness of the object-oriented paradigm. In PROOF, each object is an instance of a class, and can be either *passive* or *active*. A *passive object* acts like a service agency. It waits passively until one of its methods is invoked by some other objects. A passive object may in turn invoke methods in other objects. An *active object* is active initially, and it may remain active throughout its execution, except for occasional suspensions for the purpose of synchronization with other objects. A *body* will be attached to each active object. A *class* is a template for a set of objects bearing similar behavior, and it is defined as a *generic abstract data type*. A class is defined by its interface and definition. The *class interface* describes the types of the methods provided by the class. The *class definition* consists of the composition of its local data and the definition of each of its methods, which are purely applicative functions. Methods are defined as purely applicative functions or functional forms, i.e., high-order functions. We use a *constructor* $[x_1, x_2, \dots, x_n]$ to denote a sequence of homogeneous or heterogeneous elements. In the case of homogeneous elements, it denotes a *list* or an *array* whose types are T^* and $T^n (\equiv \underbrace{T \times T \times \dots \times T}_n)$ respectively. In the case of heterogeneous elements, it denotes a *Cartesian product* whose type is $\prod_{i=1}^n T_i (\equiv T_1 \times T_2 \times \dots \times T_n)$.

PROOF assumes that there is a set of primitive functions and functional forms from which other functions and functional forms can be easily constructed. The following are some of the functions and functional forms in PROOF.

a) Functional form: α (called *apply to all*)

Type: $(T_1 \rightarrow T_2) \rightarrow T_1^* \rightarrow T_2^*$

$\alpha f[x_1, x_2, \dots, x_n] \equiv [f(x_1), f(x_2), \dots, f(x_n)]$

α has two parameters, a function of type $T_1 \rightarrow T_2$, and a list of homogeneous elements of type T_1 . The function f is applied to each element in the list and yields a list of elements of type T_2 .

b) Functional form: β (called *distributed apply*)

$$\text{Type: } \prod_{i=1}^n (T_i^{(1)} \rightarrow T_i^{(2)}) \rightarrow \prod_{i=1}^n T_i^{(1)} \rightarrow \prod_{i=1}^n T_i^{(2)}$$

$$\beta[f_1, f_2, \dots, f_n][x_1, x_2, \dots, x_n] \equiv [f_1(x_1), f_2(x_2), \dots, f_n(x_n)]$$

β has two parameters, a list of functions in which each function f_i is of type $T_i^{(1)} \rightarrow T_i^{(2)}$, and a list of heterogeneous elements in which the i th element is of type $T_i^{(1)}$. Each function in the first list is applied to the corresponding element in the second list. It yields a list in which the i th element is of type $T_i^{(2)}$.

c) Function γ (called *filter*)

$$\text{Type: } \text{bool}^n \rightarrow T^n \rightarrow T^k, 0 \leq k \leq n$$

$$\begin{aligned} & \gamma[b_1, b_2, \dots, b_n][x_1, x_2, \dots, x_n] \\ &= \begin{cases} [], & \text{if } n = 1 \text{ and } b_1 = \text{False} \\ [x_1], & \text{if } n = 1 \text{ and } b_1 = \text{True} \\ \gamma[b_1 \dots b_k][x_1 \dots x_k] \circ \gamma[b_{k+1} \dots b_n][x_{k+1} \dots x_n], & \text{if } n > 1 \end{cases} \end{aligned}$$

Here, \circ denotes the concatenation of two lists. It is written in infix form for the sake of readability. γ has two parameters, a list of booleans and a list of any elements. This function yields a subsequence of the second list by selecting elements whose corresponding elements in the first list are True.

Both inheritance and genericness are supported in PROOF. Inheritance is used to define a subclass as a specialization of a superclass. In a subclass, all the local data and the methods of its superclass are inherited. Additional local data, new methods may be introduced. The inherited methods may also be overridden by a new definition of the method.

In PROOF, the synchronization among the objects is achieved by attaching an optional precondition, called *guard*, to each of the methods in a class. Each guard is a predicate. The object which invokes the method is suspended when the attached guard evaluates to False, and it is resumed when the guard becomes True. The guard attached to a method is defined in a way that it only depends on the status of the local data, and does not depend on the definition of any other methods.

A major deficiency of the functional paradigm is its history-insensitivity. PROOF is made history sensitive by making the objects persistent and allowing the reception of values by objects, i.e., the assignment of values to objects. The local data of objects is persistent. The reception of values by objects will modify the local data of objects. A pseudo-function \mathcal{R} , called the *reception function*, is introduced to denote the reception of a value by an object.

$$\mathcal{R}[O](e)$$

\mathcal{R} is not a function, but can be treated as a function. \mathcal{R} has two parameters: an object O , the recipient, and the expression e , to be received by O . Expression e

Table 1: A multi-mode locking mechanism.

	R-Lock	W-Lock	M-Lock
R-Lock	compatible	compatible	incompatible
W-Lock	compatible	incompatible	incompatible
M-Lock	incompatible	incompatible	incompatible

may contain applications of applicative functions only. This pseudo-function can only appear inside the bodies of active objects, and may not be nested. Major differences between modification of objects through \mathcal{R} and traditional assignments are :

- The evaluation of the expression e in \mathcal{R} can be parallel since e contains only applications of purely applicative functions.
- No partial modification to the object O is allowed. The local data of an object can only be modified as a whole entity, i.e., its components cannot be modified individually.

In PROOF, an object can simultaneously participate in more than one function evaluation. This implies that there can be more than one attempt to modify the same object simultaneously. Simultaneous modification of objects will result in inconsistent and incorrect states of objects. It is imperative that at any moment an object can be the recipient of only one of the function evaluations in which the object is involved. Simultaneous modification to the same object must be serialized. At any moment, the status of any object involved in an expression falls into one of the following three categories:

read-only : The expression only needs to read the value of the object.

will-modify : The expression will modify the object, but the modification does not occur at this moment.

modifying : The expression is currently modifying the object.

In order to ensure the consistency and correctness of objects, a multi-mode locking mechanism is adopted. There are three different types of locks, R-Lock, W-Lock and M-Lock that are associated with the three status of the object, read-only, will-modify and modifying, respectively. A lock is granted only when it is compatible with other locks granted for the same object, according to the compatibility chart in Table 1.

The programming language based on PROOF is called PROOF/L [1]. A PROOF/L program consists of a set of objects, and its methods are written based on the functional paradigm. Programs, written in PROOF/L will liberate the programmer from the burden of concerning synchronization, parallelization and communication while programming and will also make the design of the software system independent of

the parallel processing system architecture [3, 4] on which the software is to be implemented. As the programmer is liberated from the burden of concerning synchronization, parallelization, and communication these tasks are performed by the translator which translates the PROOF/L code into any target language. This makes the software written in PROOF/L portable and easy to develop. The concrete syntax of PROOF/L is given in Section 5.1.

Chapter 3

Overall Framework

As mentioned before, our approach to software development for parallel processing system is based on the computation model PROOF [1, 2] which incorporates the functional paradigm into the object-oriented paradigm. The object-oriented paradigm reflects the parallel structure of the problem space and is suitable for representing inherently concurrent behavior, and the functional paradigm allows us to exploit the parallelism in each object. Our framework consists of the following phases: object-oriented analysis, object design, verification, coding, transformation from PROOF/L to any target language of a parallel processing system as shown in Figure 3.1. In this chapter, we will give an overview of our framework.

3.1 Object-oriented Analysis

In the object-oriented software development, there is no clear distinction between the software requirement analysis phase and the system design phase since the elements of interest in each phase are still the same: the objects in the system. Thus, we do not make any distinction between the two phases and we call the first phase of our framework as object-oriented analysis. The object-oriented analysis ¹ phase consists of the following steps.

- 1) Identify *objects* and *classes*.
- 2) Determine *class interfaces*.
- 3) Specify *dependency* and *communication relationships* among objects.
- 4) Identify *active*, *passive* and *pseudo-active* objects.
- 5) Identify the *shared* objects.
- 6) Specify the behavior of each of the objects.

¹We also call the object-oriented analysis in our framework as decomposition because the system is decomposed into a set of objects in this phase.

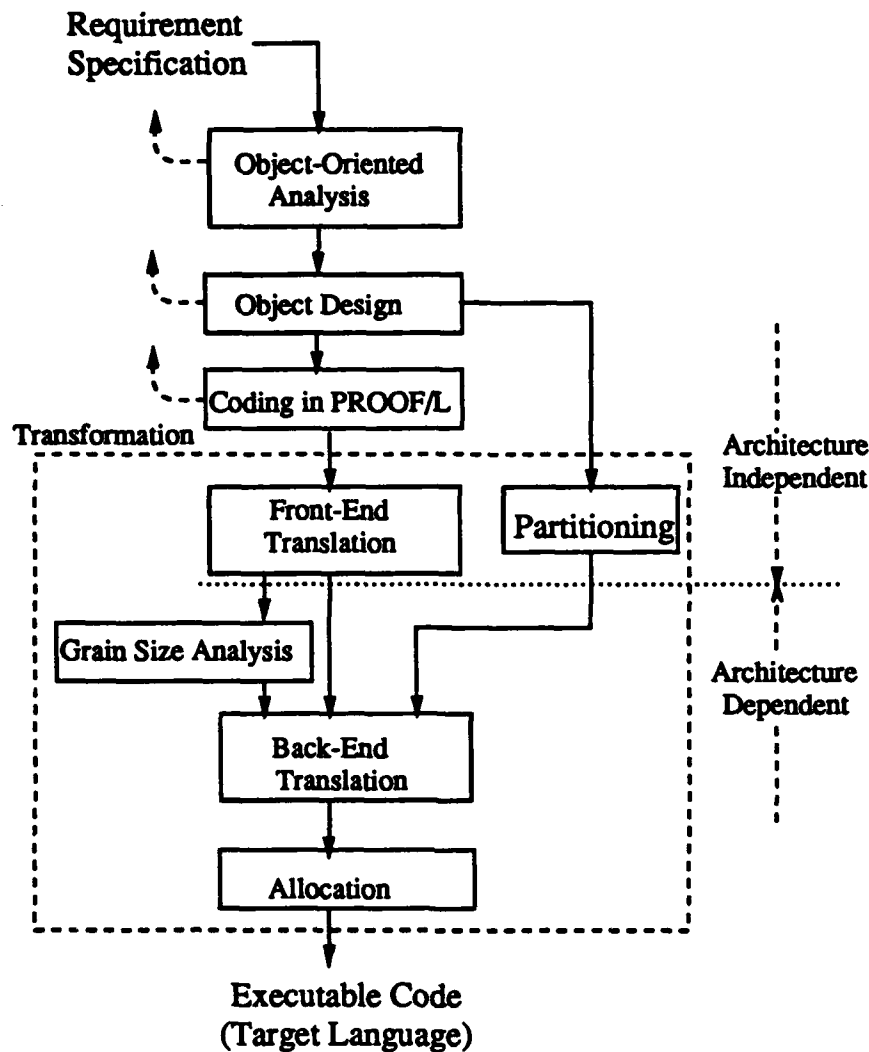


Figure 3.1: The various phases of our framework.

- 7) Identify bottleneck objects, if any.
- 8) Check the completeness and consistency.

3.1.1 Identifying Objects and Classes

In this step, the software system is represented by a set of communicating objects. Objects are identified by analyzing the semantic contents of the requirement specifications. All physical and logical entities are recognized. Each object corresponds to a real-world entity, such as sensors, control devices, data and actions. Objects having common behavior are grouped together to form a class hierarchy. The identification of the objects is currently based more on the intuition of the developer. One of the strategies to identify the objects is by examining the specification written in natural

languages. The nouns in the specification can be the candidates for the objects and the verbs for the operations [21]. Another strategy is to draw the dataflow diagrams first and then detect the candidates for objects from this diagram [22]. In the data flow diagrams, the function names are represented in the format of *action_object*, i.e., the first part of the function name denotes action and the second part of the function name denotes object. Other techniques for identifying the objects are discussed in [21]. These techniques can be used as guidelines for identifying objects. However, the experience and intuition of the programmer play an important role in identifying the objects from the requirement specifications.

3.1.2 Determining Class Interfaces

In this step, object class interfaces are determined. Because every object is considered as an instance of an object class, instead of defining objects directly, the object classes to which they belong must be defined. The interface of an object class consists of the specifications of the methods provided by the class. For each method, its specification consists of the input and output parameters and their types. The actual definitions of the methods are hidden and will be defined in a later stage. The class interface definition in PROOF is slightly different from that in the conventional object-oriented approach. Let m be a method of class C . In conventional object-oriented approach, the specification of m may appear as follows: $m : I \mapsto O$, where I is the input parameter(s) of m and O the output parameter(s) of m . Typically, m will also have side-effects on the internal states S of the instances of C . In PROOF, the methods are defined as applicative functions. Therefore, no side-effects will occur. The internal state of an object will be an explicit input and/or output parameter of m if the internal state is accessed and/or modified. Typically, in PROOF the interface of m appears as follows:

$$m : I \times S \mapsto O \times S.$$

The methods will not directly modify the state of the objects. Instead, a new state of an object will be returned when the object needs to be modified. The modification of objects will be achieved by a special construct discussed later.

3.1.3 Specifying Dependency and Communication Relationships Among Objects

In this step, the static relationships among objects are specified using the object communication diagrams. The identity of the objects, the methods in each object and the relationships among them are specified to capture the features of the real world problem which are important for the software developer. In the object communication diagram, the objects are represented as rectangles. The links between the objects indicate the communication between objects, i.e., method invocation. The arrows on the links indicate the directions of invocation. The methods defined in an object as

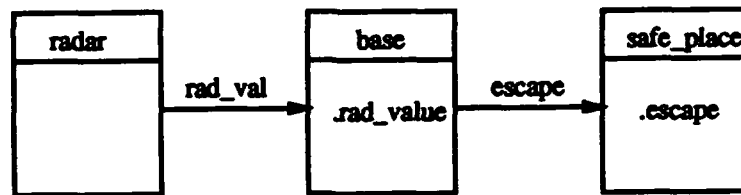


Figure 3.2: An object communication diagram.

interface are written within the object with the method names beginning with a period. The labels on the arrows show which methods are being invoked by an object. Figure 3.2 shows an example of an object communication diagram, where the objects *radar*, *base* and *safe_place* interact with each other by invoking their methods. Object *radar* invokes the method *rad_val* defined in object *base* and *base* invokes the method *escape* defined in the object *safe_place*. A more elaborate object communication diagram will be given when we present an example to illustrate our approach in Chapter 8.

3.1.4 Identifying Active, Passive and Pseudo-Active Objects

In this step, the objects are classified according to their invocation properties as *active*, *passive* or *pseudo-active*. An *active object* can initiate activation of other objects by invoking methods of other objects. The methods defined in an active object cannot be invoked by other objects, but they can be invoked by other methods defined within the active object itself. A *passive object* is activated only when its methods are invoked by other objects. *Pseudo-active objects* behave between the active and passive objects. Pseudo-active objects can invoke the methods of other passive or pseudo-active objects and also has methods which can be invoked by other active or pseudo-active objects. Since active objects are invoked when the software system is started, all the threads of control in the application start from the active objects. Identifying all the threads is very important in real-time process control systems because we need this information to check for the completeness and the consistency of the decomposition. We can easily identify the active, passive and pseudo-active objects from the object-communication diagram. The active objects have only outgoing arrows, the passive objects have only incoming arrows and the pseudo-active objects have both incoming and outgoing arrows. Classification of objects by their invocation behavior helps us build the static structure of the software system among objects.

3.1.5 Identifying Shared Objects

In this step, once the static structure of the software system is determined, we identify the shared objects from them. An object is a *shared object* if it has local data which can be accessed by a number of objects. The shared objects can be further divided into two classes of objects: *read-only* shared object and *writable* shared object. The read-only

object has local data which cannot be modified by other objects. The writable object has local data which can be modified by other objects. Read-only objects can be freely duplicated as many times as desired. However, writable objects cannot be duplicated easily because maintaining the consistency of the data will then become an overhead. All the access to the data in the writable shared objects needs to be synchronized to maintain the consistent status of the data. An active object cannot be a shared object, since by definition, no other object can invoke the methods in an active object. Shared writable objects could become bottleneck objects as they may have to be executed sequentially to maintain the consistency of the data. Such bottleneck objects are often shared components requiring synchronization among objects accessing it concurrently. Identifying such bottleneck objects from the decomposition and refining the decomposition to reduce or remove some of such unnecessary bottleneck objects play an important role in enhancing the parallelism.

3.1.6 Specifying the Behavior of Objects

In this step, the behavior of each object is specified. The object communication diagram obtained in Step 3) only describes the static structure and relationships of the objects in the problem domain. It does not provide any information regarding the behavior of the software system to be developed. That is, the control aspect of the software system is not specified in the object communication diagram. However, to verify and analyze the decomposition, we need to define the behavior of each object. For this purpose, we use the notations similar to those in [23]:

- SEQuential execution of methods: When the methods m_1, m_2, \dots, m_n are executed sequentially in the order m_1, m_2, \dots, m_n , its behavior is specified as:

$$\text{SEQ}(m_1, m_2, \dots, m_n)$$

- CONcurrent execution of methods: When the methods m_1, m_2, \dots, m_n are executed concurrently, its behavior is specified as:

$$\text{CON}(m_1, m_2, \dots, m_n)$$

- WAIT for method invocation: When an object is waiting for the invocation of its method m by another object O to proceed with its execution, its behavior is specified as:

$$\text{WAIT}(m, O)$$

- SElect a method for execution based on a condition: SEL construct behaves like the CASE statement in ordinary programming languages. The SEL construct selects one of the methods based on a condition. When an object selects one of the methods for execution from the methods m_1, m_2, \dots, m_n based on a condition C , its behavior is specified as:

$$\text{SEL}(C; m_1, m_2, \dots, m_n)$$

- **ONE-OF** the methods for execution from a group of possible methods: **ONE-OF** construct is used in cases where different objects could try to invoke the methods defined in the object O simultaneously, but the O permits only one object to invoke its method at a time. That is, this construct serializes the requests, and it is typically used to describe the behavior of shared writable objects. Note the difference between the **SEL** and the **ONE-OF** construct. Among the set of methods m_1, \dots, m_n defined in an object when the object permits only one of its methods to be invoked by other objects, the behavior of the object is specified as:

$$\text{ONE} - \text{OF}(\text{WAIT}(m_1, O_j), \dots, \text{WAIT}(m_n, O_k))$$

ONE-OF will always be associated with a **WAIT** construct in a shared writable object because the object O will have to wait for other objects to invoke its methods.

3.1.7 Identifying Bottleneck Objects

In this step, the bottleneck object which may unnecessarily degrade the performance of the software system is identified. Usually, a bottleneck object will be a shared writable object. One can identify a shared writable object from the description of the object behavior in the above step. If the behavior of an object has a construct of the type $\text{ONE} - \text{OF}(\text{WAIT}(\dots), \text{WAIT}(\dots)\dots)$, then this object is also a bottleneck object. Such objects limit the parallelism in the software system. If such an object is found, then redo or refine the object-oriented analysis to reduce the bottleneck if possible. This step may increase the number of objects in the software system. Repeat Steps 2) to 6) until the object-oriented analysis is found satisfactory.

3.1.8 Checking For Completeness and Consistency

In this step, the result of the object-oriented analysis is verified with the user requirements. From the given user requirements, the possible threads of controls are identified, and each of them is examined using the behavior of the objects specified in Step 5). The first activity in any control thread must begin in one of the active objects. The sequence of activities in each control thread must be reachable by tracing the behavior of the objects. If there is any control thread that cannot be followed, the decomposition is incorrect and the decomposition steps need to be reviewed. The consistency among objects is verified by examining whether input parameters of the methods being called are defined as local variables in the calling object and the output parameters of the methods being called are defined as local variables in the called object.

3.2 Object Design

In our approach, the object design is specified using the notations defined in PROOF/L [1]. The class interface definitions and information about the object behavior are used to design the objects. Our approach to object design involves three steps:

- 1) Establish the class hierarchy.
- 2) Design the class composition and the methods in each object.
- 3) Design the bodies of the active and pseudo-active objects.

3.2.1 Establishing Class Hierarchy

Since some common operations and/or attributes between the objects may not be apparent in the analysis phase, different objects are reexamined to identify the commonality among the classes in the design phase. A set of operations and/or attributes that are common to more than one class can be abstracted and implemented in a common class called the *superclass*. The subclasses then have only the specialized features. In some cases, a superclass can be extracted from a single subclass and put in the class library if needed. Establishing a class hierarchy in the form of superclasses and subclasses increases the inheritance in the application. Class hierarchy also enhances the modularity and the extensibility of the software system [24].

3.2.2 Designing Class Composition and Methods

In this step, the composition and the methods for each object class are designed. The class definition consists of *composition* and *methods*. The composition defines the internal data structure of the class. Various constructors, such as list and Cartesian product, are provided. A typical functional style is adopted in the method definition. A rich set of functional forms, i.e. high-order functions, as well as primitive functions are predefined. In the method design, the internal state of the object to which the method belongs is included as both the input and output parameters so that side-effects can be avoided. A method of an object consists of an optional *guard* and an *expression*. The guard is a predicate specifying synchronization constraints and the expression statement specifies the behavior of the method. The synchronization among concurrent objects is achieved by the guards attached to the methods. The guard attached to a method is defined in a way that it only depends on the status of the local data, but does not depend on the definition of other methods. Therefore, the guards are directly inheritable with the methods. The expression is a purely applicative function and is specified informally in a natural language. Due to the referential transparency of applicative functions, fine grain parallelism can be exploited. The method execution can be done as follows:

1. Evaluate the guard associated with the method. If the method does not have a guard, then we assume that we have a guard that evaluates to *true* always.
2. If the guard is *true*, then execute the expression; otherwise, go back to (1).

When there are simultaneous attempts to access the same object through invocation of its methods, the selection of one method for execution is done non-deterministically.

It is desirable to refine the methods that access the shared objects. For example, let object *O1* invoke a method *m* defined in the shared object *O2*. Now suppose that the method *m* requires to read data, perform some computation based on the data and then modify the local data of *O2*. Then the guard of the method *m* needs to be evaluated before the execution of the method *m* begins. The activities, reading and computing, performed on *O2* can be executed in parallel when another object invokes this method because those operations do not involve any shared data. However, these activities cannot be invoked by another object in parallel if the method *m* contains these activities as part of its code. Thus, the method should be refined into smaller methods in such a way that the guard can affect the execution of a short segment of code only. This refinement of method is similar to the refinement of the object to reduce the bottleneck in the object-oriented analysis stage.

Selection of algorithm and data structure is an important part of the method design. The selection of algorithms to accomplish a specific task should be based on certain criteria which satisfy the required constraints such as accuracy, timing requirements, use of common utilities across the design, reuse of previously developed software, computational complexity, flexibility, ease of implementation, understandability, etc.

The underlying architecture of the machine will not be an influencing factor if the algorithm is to be executed on a single processor. On the other hand, if the algorithm is to be executed on a configuration of parallel processors, then the algorithm selection decision will usually be influenced by the underlying architecture on which the algorithm is to be executed. In the case of reconfigurable architectures, the algorithm selection can be based on the performance requirements and the designer will not need to be concerned with the configuration of the computer. The computer can then be reconfigured to reflect the structure of the algorithm. The designer of the algorithm to be executed on more than one processor on a parallel processing system should be aware of the configurations that are available on the system, such as the maximum number of immediate neighbors that a processor could have in the system architecture, etc. For example, a transputer could have a maximum of four immediate neighbors; and a hypercube could have a number of immediate neighbors depending on the dimension of the hypercube. While designing the algorithms, new classes of objects may be defined to make the implementation more efficient. These are low level objects and are not usually visible externally. New classes called *internal classes* may also be defined at this stage for the purpose of implementation, but they are not reflected in the user requirement.

3.2.3 Designing the Bodies for Active and Pseudo-Active Objects

In this step, a body is associated with each active and pseudo-active object. There is no body associated with a passive object as it does not invoke any methods. The role of a body is to invoke a method and to modify the state of the objects represented by their local data. The body in each object is expressed in the form $e_1//e_2//\dots//e_k$ where each e_i is an expression representing method invocations and expressions separated by $//$ are evaluated simultaneously. $//$ is a parallel construct indicating parallel execution. f_i can be recursively defined and can be diverse. Thus, the evaluation process may be infinite. In f_i , methods of objects can be invoked, and the states of objects may be modified. The modification of an object is expressed by the *reception* construct which has the form $R[|O|]e$, where O called a *recipient object* is an object name and e is an expression with applications of purely applicative functions only. The reception construct can occur only in the bodies of active and pseudo-active objects. The reception construct indicates that the object O will receive the value returned as a result of evaluating the expression e . This construct modifies the states of the object. It differs from the conventional assignment in the following aspects:

1. The expression e contains only purely applicative functions. Thus, the evaluation of the expression e is side-effect free and can be parallelized.
2. The expression e must return a new state of the object O . O receives the new state as a whole entity. Therefore, no partial modification and no inconsistent state of the object is possible.

The object O may be composed of other objects. However, the composing objects cannot appear in the reception construct as a recipient. The overhead involved in complying with this no partial modification rule can be minimized by optimization based on static data dependency analysis.

The body of an object can be derived using the class interface and the object behavior obtained from the analysis stage. We also need to introduce the modification operator \mathcal{R} in the body of the objects that are modified. The objects that are modified can be determined from the method definitions given in the class interface. Consider an object O_1 defined as the output of a method m . Whenever an object O_2 invokes the method m defined in O_1 , O_2 will be modified. Thus, in the body of O_2 , when m is invoked, the modification operator $\mathcal{R}[|O_2|]m$ is substituted in the place of the method invocation.

The object behavior is specified using the control constructs, SEQ, CON, SEL and ONE-OF, and statements including method names and WAIT clauses. Transforming the control constructs into equivalent body is straightforward. The SEQ, CON and SEL constructs are transformed into sequential, concurrent, and if-then-else types of constructs respectively in the body. These constructs appear in the active, passive and pseudo-active objects. WAIT construct appears only in the passive and pseudo-active objects, and not in the active objects as the active objects do not wait for other

objects to invoke their methods. ONE-OF construct appears only in the passive objects or in the passive part of the pseudo-active objects. A WAIT construct is usually associated with the ONE-OF construct.

When a WAIT construct is encountered in the object behavior, then a guard has to be associated with the method which is *WAITing* to be invoked by other objects. The guard value will be set/reset by the interacting objects. Alternately, a guard can also be introduced into the body of the object which is *WAITing*. However, since the computation model PROOF supports only the former, we will use only the former strategy. The latter method has the advantage in that the body design follows directly from the object behavior, while in the former, the methods will have to be designed only after taking the WAIT constructs into consideration. The design of the method will then become dependent on the object behavior, introducing an extra level of complexity which can be avoided when a guard is introduced at the body level.

The behavior of a purely passive object begins with a WAIT clause or a ONE-OF clause. The guard mechanism in this case can be used to enforce mutual exclusion if the object is a shared writable object. A pseudo-active object should also begin with a WAIT clause. The method *WAITing* to be invoked should then be redesigned to include a guard which will be set only by the invoking object and will be reset by the called object. In such a case, the invoking object will then clear the way for the pseudo-active object to start its thread of operation. The guard will have to be checked by the pseudo-active object and continue further execution only when the guard evaluates to True. For this purpose, we can add an additional method in the class of the pseudo-active object which will only evaluate this guard. This step is further illustrated in the example, discussed later, when designing the body of the pseudo-active object. If the analysis was such that a pseudo-active object starts its thread in the beginning and then encounters a WAIT state, then we can break up this object into an active object and a passive object, with an object communication link between these two objects.

3.3 Verification

The design of the objects done in the previous phase has to be verified and analyzed for various *liveness* and *safeness* properties. For this purpose, we transform our design into Petri nets [25]. Petri nets have been selected in our approach mainly because our design can be easily represented in the Petri-net model and because many techniques have been developed to analyze Petri-net models for various liveness and safeness properties [26]–[32]. An extensive survey of Petri nets and their applications is given in [33], and an overview of existing tools is given in [34].

Petri nets can be used to model both the static and dynamic properties of the systems. Static properties of the systems are represented by the graph part of a Petri net, while the dynamic properties of the system can be determined from the Petri-net graph, the initial marking and the firing rules. The advantages of modeling a

dynamic system with Petri nets are: their ability for graphical and precise nature of representation; analysis tools to determine and verify the dynamic behavior of the system from its structure; and the capability to design the system using top-down and bottom-up approaches.

The transformation of our design to Petri nets consists of the following three steps:

- 1) Transformation of bodies to Petri nets.
- 2) Composition of the Petri nets.
- 3) Refinement of the Petri nets.

3.3.1 Transformation of bodies to Petri nets

To transform the bodies designed into Petri nets, we use places as the token holder for the control flow, transitions as the methods, and the arcs between places and transitions as the control flows. Since a body is represented as a statement consisting of control constructs and method names, we show the transformation for each of control constructs, viz., CON, SEQ, SEL and ONE-OF in Figure 3.3. Expressions in the body of an active object could have methods that do not require modifications and/or methods that require modification by using the construct \mathcal{R} . The method requiring modification needs to be executed in serial to maintain the consistency of the object state. These two kinds of methods are represented differently in the Petri-net representation after the transformation. For this purpose, an additional place called the *bottleneck* place is associated with such a method. The bottleneck place serializes the execution of the methods. Figure 3.4. shows the transformation of such a method. The bottleneck place will also be used to compose the Petri nets in the next step.

3.3.2 Composition of the Petri-nets

The Petri-net representations of the bodies of the active or pseudo-active objects that interact with each other should be composed together so that they can be analyzed together. To compose the nets, we identify the transitions or the places that serve as interaction points. The interaction among objects occurs only when there is an object modified by other objects. When an object interacts with another object for accessing the shared writable object, the bottleneck place will be common to both the objects. Since the bottleneck place is used to serialize the interaction among the methods requiring modification, they can be used as the fusion point. When the nets are to be composed, the body of the active object is searched for the methods that require modification. When such methods are found, two cases arise. For example, consider two active objects O_A and O_B having the following bodies:

$$O_A: \text{SEQ}(m_1, \dots, \mathcal{R}[O_i] m_n)$$

$$O_B: \text{SEQ}(m_1', \dots, \mathcal{R}[O_j] m_{n'})$$

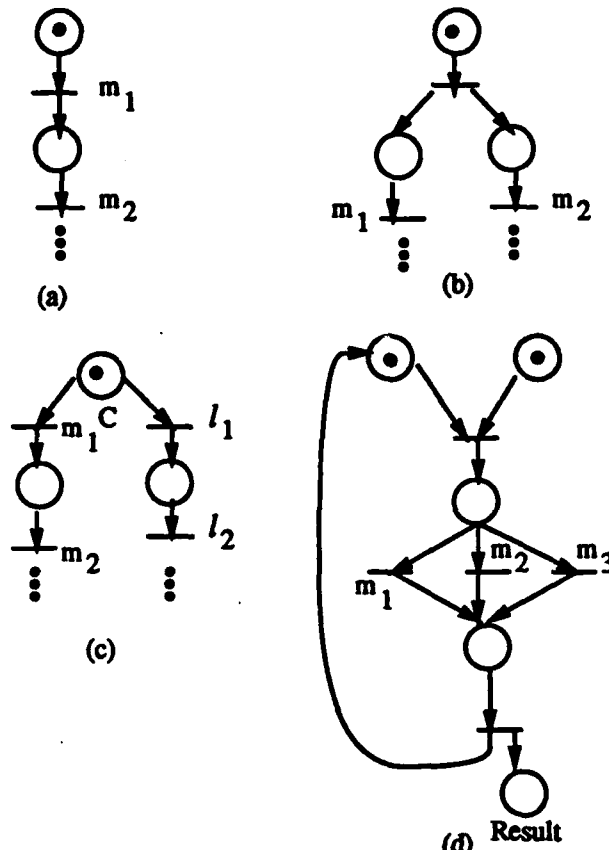


Figure 3.3: Transformation of the control constructs: (a) $SEQ(m_1, m_2, \dots)$, (b) $CON(SEQ(m_1, \dots), SEQ(m_2, \dots))$, (c) $SEL(C; SEQ(m_1, m_2, \dots), SEQ(l_1, l_2, \dots))$, (d) $ONE-OF(m_1, m_2, m_3)$.

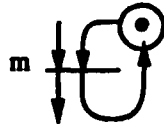


Figure 3.4: Transformation of a method requiring modification.

When the methods m_n and $m_{n'}$ are defined in different classes ($O_i \neq O_j$) there is no common bottleneck place between O_A and O_B . Hence, no composition of the nets is necessary. When the methods m_n and $m_{n'}$ are defined in the same object ($O_i = O_j$), the two bottleneck places associated with the two methods are combined to one. This process is called *fusion* of places and is illustrated in Figure 3.5.

3.3.3 Refinement of Petri nets

The purpose of the refinement is to replace a transition or place by a more complex Petri net in order to give a more detailed description of the activity involved in the transition or place respectively. It is analogous to the module concepts found in many programming languages. At one level, a simple abstract description of the activity is given without considering the detailed behavior. At another level, by refining the

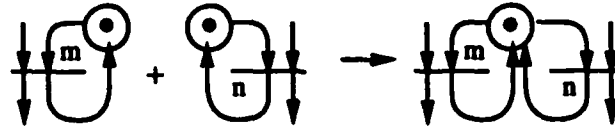


Figure 3.5: Combining two objects with a common bottleneck place.

nets, a more detailed description of the activities taking place at the transition or place is specified.

The transition/place can be refined according to the following rules. Suppose that a transition/place t_i is replaced with a subnet S .

- The subnet S consists of three parts: input transition/place, a refinement of net called block, and output transition/place.
- The incoming arcs of t_i serve as the incoming arcs to the input transition/place.
- The outgoing arcs from t_i serve as the outgoing arcs from the output transition/place.
- Only one transition receives all the input parameters.
- Only one transition produces all the output parameters.
- All the transitions except these two input and output transitions can only interact with the places and transitions defined within the subnet S .
- All the places can only interact with transitions defined within the subnet S .

For example, since a method consists of an expression with an optional guard, the transitions may have to be refined to specify the guard and the expression. This is done as follows: Let a method m_i consist of a guard g_i and an expression e_i . Then the transition for m_i can be refined as follows: Guard evaluation is specified as a place and there is a transition associated with each result - true or false - of the guard evaluation. In case of True transition, the expressions are executed. In case of False transition, go back to the guard place to evaluate the guard again. The use of True and False transitions is analogous to the method specified in [25] to represent condition statement. This refinement process is shown in Figure 3.6.

After the coding, when the complete definition of the expressions for all methods is given, the refinement of the transitions representing all the expressions can be done. Suppose we have an expression e_i : $f(g(a,b),h(c,d))$. Then, the transition for the expression e_i may be refined as shown in Figure 3.7.

3.4 Coding in PROOF/L

The design of the software system will be implemented by writing the program in PROOF/L. The coding in PROOF/L is rather straightforward and will not be discussed. The concrete syntax of the PROOF/L language is given in Section 5.1. The

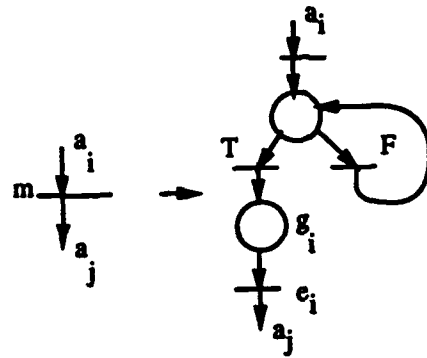


Figure 3.6: Refinement of the methods with the guards.

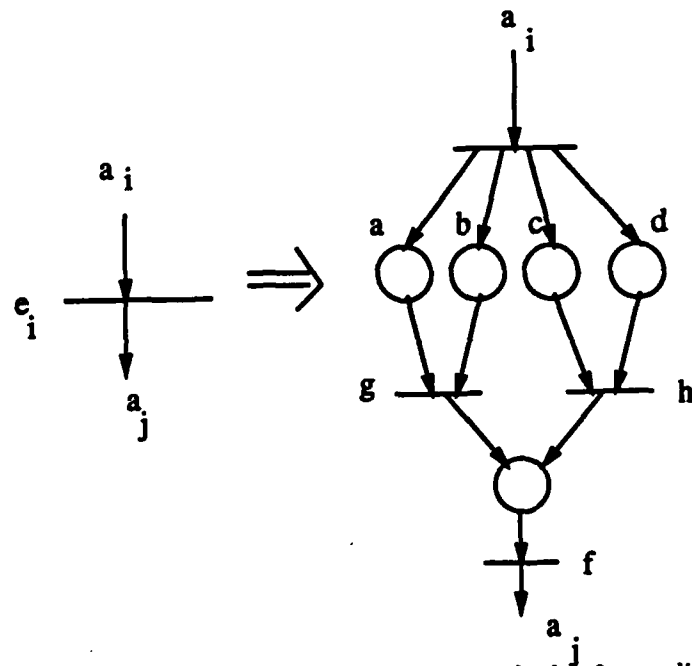


Figure 3.7: Refinement of the methods after coding.

PROOF/L code will have to be translated to the code in a selected target language that can be executed on the parallel processing system. In our case, we use a Parsytech board consisting of a network of sixteen T800 transputers, and have developed the transformation from PROOF/L code to INMOS C code.

3.5 Transformation from PROOF/L to any Target Language

The transformation of the program written in PROOF/L into a target language involves the five major steps as shown in Figure 3.1: partitioning, front-end translation, grain size analysis, back-end translation and allocation. In the partitioning step, the objects in the software system are partitioned into a set of clusters. The objective of our partitioning approach is to improve the performance of the software by reducing communication cost among processors while maintaining the parallelization among objects. During this phase, we are only concerned with the coarse grain parallelism among objects. In the front-end translation, PROOF/L code is translated into an Intermediate Program Representation (IPR). The purpose of this phase is to express all the parallelism in the PROOF/L code explicitly. As discussed in Section 2.2, functional paradigm in PROOF makes it easy to detect all the parallelism. Once the parallelism is expressed explicitly, grain size analysis is performed to determine the proper size of tasks to be executed in different processors. For this purpose, IPR is served as a task precedence graph and a modified intermediate form is generated. In the back-end translation, the modified intermediate form is translated into corresponding equivalent Inmos C code. Then the Inmos C code is allocated to physical processors. In Chapters 4 through 7, we will present our techniques for partitioning, front-end translation, grain size determination and back-end translation in detail.

Chapter 4

Partitioning

In order to distribute the modules to processors of a parallel processing system so that the execution time of the software can be minimized, we need to partition the software system into a set of modules and then assign them to the processors. We call the first stage *partitioning* and the second *allocation*. Intuitively, to exploit parallel processing power, the modules should be executed in parallel as much as possible. On the other hand, to reduce the negative effect of high communication overhead on the system performance, the modules should be distributed over as few processors as possible. The trade-off between these two conflicting criteria has been well known in the study of parallel processing systems as well as distributed computing systems. In parallel processing systems, it is very difficult to achieve linear speedup due to communication costs among processors, contention of shared resources and inability to keep all the processors busy as much as possible [36]. That is one of the reasons that there is a large gap between the ideal peak performance and the real performance in most parallel processing systems.

The problem of partitioning for parallel and distributed computing systems has been studied extensively. The approaches are divided into three categories: graph-theoretic [37, 38], integer programming [39, 40] and heuristics [41]-[45]. The graph theoretical approach [37] uses an undirected graph to model a software system, and then minimizes the total interprocess communication cost by performing a min-cut algorithm on the graph. Its application is limited due to its exponential time complexity as more than two processors are used. Furthermore, the incorporation of various constraints into such a model is difficult. The integer programming approach [39, 40] is based on the implicit enumeration algorithm. It is easy to incorporate additional constraints into their models in order to satisfy various application requirements. But, the amount of time and memory space required to obtain an optimal solution grows exponentially with the number of modules of the software system. The above mentioned approaches attempt to find partitions with the objective of minimizing the sum of the communication time and the processing time of the software. In general, this problem is NP-hard. Thus, heuristic approaches are applied to provide fast and effective algorithms for a suboptimum solution. Comparing with optimal solution methods, the heuristic methods are faster, more extensible, and simpler. They are

also applicable to large dimensional problems and the problems for which optimal solutions cannot be obtained in real time.

One of the common assumptions in these approaches is that the execution time for each module and the communication time among modules are given as input. In other partitioning approaches [46]-[48], the software is represented as a directed graph in which each node represents a computation task and each arc between two nodes represents precedence relation in terms of data flow or control flow. However, these approaches cannot be directly applied to the partitioning stage of the object-oriented software development for parallel processing systems because they ignore the existence of the shared data. For instance, in our approach, the software system is considered consisting of a set of objects where every object can contain shared data that may be accessed by a number of objects. When the access to the shared data requires modification of the data, the access must be serialized in order to maintain the consistency of data state. When an object containing shared data is simultaneously accessed by a number of objects and the accesses do not require modifying the shared data, the parallel invocation of methods in the object should be allowed. Most of the existing partitioning approaches cannot be used when the software is decomposed as a set of such objects. In this section, we will present a partitioning approach to overcome these difficulties.

4.1 Our Partitioning Approach

The objective of our partitioning approach is to improve the overall performance of the software by reducing communication cost among processors while maintaining the potential parallelism among objects. The details of our partitioning approach with illustrative examples has been presented in [49].

The input for our algorithm includes the behavior of the objects in the software system that is expressed using the constructs discussed in Section 3.1.6, communication intensity information extracted from the requirement analysis, and the number of replications for each object as required for such purposes as fault tolerance. The output of our algorithm will be an undirected weighted graph in which every node represents a cluster of objects and every edge between two nodes has a positive weight which represents the degree of contribution that can be made to the enhancement of the overall performance by parallel execution of the two clusters represented by the two nodes. Our partitioning approach consists of the three parts: initialization, normalization and clustering.

4.1.1 Initialization

We begin with an undirected weighted graph, in which each node represents an object, and there is an edge between the nodes of two objects if and only if either one of the two objects invokes the other or both objects are invoked concurrently by another

object. Every node has a non-negative weight that is equal to the number of the replications of the object represented by that node. Every edge in the graph has two kinds of weights: one for communication and another for concurrency.

Communication weight associated with an edge represents the communication cost incurred if the objects of the two nodes incident to the edge communicate with one another, and they are not allocated on the same processor. If object O_i invokes another object O_j (i.e. there is a communication overhead between the two objects), then a non-zero communication weight u_{ij} is assigned to the edge connecting the nodes of those objects. The value of u_{ij} is equal to the product of object invocation frequency, f_{ij} , and the number of data units transferred between the two objects every time one of them invokes the other (i.e. the two objects communicate). We assume that the information needed to compute the communication weight can be obtained from the analysis of requirement specification. A negative sign is given to communication weights to imply cost. As a result, the smaller communication weight implies the more communication overhead.

Concurrency weight associated with an edge corresponds to the gain in improving the overall performance of the system that can be obtained by parallel execution of the two involving objects. If two objects are invoked concurrently by another object, there is a potential parallelism between the two objects; and a non-zero concurrency weight equal to the frequency by which the two objects are invoked is assigned to the edge connecting their nodes. A positive sign to concurrency weights is given to imply the gain achieved as a result of potential parallel execution of the involving objects.

The undirected weighted graph of the software system $G = (V, E)$ has a set of nodes V and a set of edges E such that:

- Object O_i is represented by node O_i in V .
- Edge (O_i, O_j) is in E if and only if O_i and O_j can communicate with one another or both objects can be invoked concurrently by another object.
- The node of object, say a , has a non-negative weight, denoted by r_a , which is equal to the number of replications of the object modeled by that node.
- An ordered set of weights (u_{ij}, v_{ij}) is associated to edge (O_i, O_j) where u_{ij} and v_{ij} are communication and concurrency weights, respectively.

Let f_{ij} be the frequency that O_i invokes O_j and d_{ij} be the number of data units transferred between O_i and O_j every time O_i invokes object O_j . Communication and concurrency weights are assigned according to the five following rules: Rules 1-4 are applied to the cases where objects are related by only one construct.

Rule 1. $O_1 : \text{CON}(O_2, O_3, \dots, O_n)$ describes a case where objects O_2, O_3, \dots, O_{n-1} , and O_n are executed concurrently after being invoked by object O_1 . It corresponds to a subgraph $G = (V, E)$ where $V = \{O_1, O_2, \dots, O_n\}$, and $E = \{(O_i, O_j), 1 \leq i < j \leq n\}$. Communication and concurrency weights are assigned to the edges in E as follows:

- 1) For $2 \leq i < j \leq n$, there are two possibilities:
 - a) If (O_i, O_j) is new, then $u_{ij} = 0$ and $v_{ij} = f_{1i} = f_{1j}$.
 - b) If (O_i, O_j) is old, then u_{ij} remains unchanged, and $v_{ij} = v_{ij} + f_{1i}$.
- 2) For $i = 1$ and $2 \leq j \leq n$, there are two possibilities:
 - a) If (O_i, O_j) is new, then $u_{ij} = -(f_{1j} \times d_{1j})$ and $v_{ij} = 0$.
 - b) If (O_i, O_j) is old, then $u_{ij} = u_{ij} - (f_{1j} \times d_{1j})$ and v_{ij} remains unchanged.

Rule 2. $O_1 : \text{SEQ}(O_2, O_3, \dots, O_n)$ describes a case where object O_1 invokes objects O_2, O_3, \dots, O_{n-1} , and O_n in a sequential order. It corresponds to a subgraph $G = (V, E)$, where $V = \{O_1, O_2, \dots, O_n\}$, and $E = \{(O_1, O_j), 2 \leq j \leq n\}$. In assigning communication and concurrency weights to the edges in E , there are two possible cases for $i = 1$ and $2 \leq j \leq n$:

- 1) If (O_i, O_j) is new, then $u_{ij} = -(f_{1j} \times d_{1j})$ and $v_{ij} = 0$.
- 2) If $d(O_i, O_j)$ is old, then $u_{ij} = u_{ij} - (f_{1j} \times d_{1j})$ and v_{ij} remains unchanged.

Rule 3. $O_1 : \text{ONE-OF}(O_2, O_3, \dots, O_n)$ and $O_1 : \text{SEL}(O_2, O_3, \dots, O_n)$ each describes a case where object O_1 invokes only one of the objects O_j 's for $2 \leq j \leq n$. The corresponding subgraph is $G = (V, E)$, where $V = \{O_1, O_2, \dots, O_n\}$, and $E = \{(O_1, O_j), 2 \leq j \leq n\}$. In assigning communication and concurrency weights to the edges in E , there are two possible cases for $i = 1$ and $2 \leq j \leq n$:

- 1) If (O_i, O_j) is new, then $u_{ij} = -(f_{1j} \times d_{1j}) / (n - 1)$ and $v_{ij} = 0$.
- 2) If (O_i, O_j) is old, then $u_{ij} = u_{ij} - (f_{1j} \times d_{1j}) / (n - 1)$ and v_{ij} remains unchanged.

As mentioned Section 3.1.6, this construct is used to represent the synchronized access of the shared data.

Rule 4. $O_i : \text{WAIT}(O_j)$ describes a case where object O_i waits to be invoked by object O_j . It corresponds to a subgraph $G = (V, E)$, where $V = \{O_i, O_j\}$, and $E = \{(O_i, O_j)\}$. There are two possibilities:

- 1) If (O_i, O_j) is new, then $u_{ij} = v_{ij} = 0$ ¹.
- 2) If (O_i, O_j) is old, then both u_{ij} and v_{ij} remain unchanged.

Rule 5 is applied to nested clauses. Before presenting Rule 5, we define the preservation of the edge relationship, denoted by E-R, between two subgraphs. Let G_A and G_B be two subgraphs defined as $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$, where $V_A = \{x_1, \dots, x_q\}$ and $V_B = \{y_1, \dots, y_r\}$ for some $q \geq 1$ and $r \geq 1$. For every x in V_A and every y in V_B , one of the following relations hold:

¹ A nonzero communication weight will be assigned to this edge when object O_j is processed.

$E-R(x, y) = \text{True}$ if there is an edge incident to x and y .

$E-R(x, y) = \text{False}$ otherwise.

Then the preservation of the edge relationship $E-R$ between the two subgraphs is defined as follows:

$$\begin{aligned} E-R(G_A, G_B) &= E-R(x_1, y_1) \\ &= E-R(x_2, y_1) \\ &\vdots \\ &= E-R(x_q, y_1) \\ &= E-R(x_1, y_2) \\ &= E-R(x_2, y_2) \\ &\vdots \\ &= E-R(x_q, y_r). \end{aligned}$$

Rule 5. It is applied when nested clauses are used to specify the object behavior. The steps are:

- 1) Modify the object behavior by substituting all the nested clauses with dummy objects.
- 2) Select and apply a rule based on the construct used. For every dummy object introduced in Step 1), do the following:
 - 2.1) Apply an appropriate rule and preserve the edge relationships with other objects.
 - 2.2) Assign communication and concurrency weights using Rules 1-4.

4.1.2 Normalization

As stated earlier, the goal of our partitioning approach is to reduce the communication cost between the processors and to exploit potential parallelism among the objects. Since these two subgoals are conflicting, it is desirable to find an optimal point at which communication costs are reasonably reduced while the parallel execution of objects are well achieved. However, when no precise information on the execution time and communication time is available, an optimal solution cannot be found. Even if such information were available, the problem of clustering that to be discussed in the next section would remain NP-hard [50].

In order to accommodate the two conflicting partitioning subgoals, we present a normalization method so that the communication and concurrency weights associated with every edge can be combined to obtain a common metric for the two kinds of weights. Let u_{min} and v_{max} be the minimum communication weight and the maximum concurrency weight, respectively. First, we replace every u_{min} with $(1/u_{min}) \times u$ and every v_{max} with $(1/v_{max}) \times v$. This brings the two types of weights to the same scale. Then, we define for every edge a new weight W , called gain, to

be $\alpha \times u + (1 - \alpha) \times v$, where the coefficient α lies in the range of $(0, 1)$, and is determined by the characteristics of the underlying parallel processing systems. To obtain an optimal α , the configuration of the parallel processing system, CPU speed and the communication unit capability are needed which may not be available at the partitioning stage. Therefore, by allowing modification of α , the performance of the software system can be adjusted.

We now have a graph in which every node represents an object (the same as in the initial graph) and every edge has only one weight, denoted by W , that represents the degree of contribution to the overall performance of the system that can be made by the parallel execution of the objects of the nodes incident to that edge. Next, we will define the function gain for the graph to be the sum of the weights of all edges in the graph excluding any weight with $+\infty$ value.

4.1.3 Clustering

In this section, we will show how to cluster the objects represented by the nodes in the graph in order to increase the total gain for the graph by taking a bottom-up approach.

Note that an edge with a positive weight suggests that parallel execution of the two involving objects will reduce the completion time of the software system. Hence, these two objects should not be in one cluster. On the other hand, an edge with a negative weight implies that the two involving objects should be placed in one cluster because parallel execution of those objects does not reduce the completion time of the system due to the communication overhead occurring if they are not executed on the same processor. If the weight is equal to zero, we choose not to cluster the involving objects for the following reason: Clustering of such objects cannot contribute in increasing the gain for the graph. Comparing a partition consisting of many small processes with one consisting of a few large processes, the partition with many small processes will provide the allocation phase with more flexibility for the purpose of load balance or growth potential [42].

The input is an undirected weighted graph $G' = (V', E')$, where $V' = \{O_1, O_2, \dots, O_p\}$ and O_i is an object for $1 \leq i \leq p$. Every node has a non-negative weight which is equal to the number of replications of the object represented by that node. Every edge $(O_i, O_j) \in E'$ has a weight W_{ij} . We define function SIZE to map every node in the graph to a positive integer that is equal to the number of objects in the cluster represented by that node. The value of function SIZE at any node in the initial graph is defined as one because every node in the initial graph represents only one object.

The steps we take in clustering the nodes of the graph are as follows:

```

Step 1. for every node  $c$  do set  $\text{SIZE}(c) = 1$ .
while there is an edge with a negative weight and
      there is more than one node in the graph do

```

```

begin
Step 2. Find the edge in the graph with minimum weight.
        Let it be edge  $(a, b)$  with a weight  $W_{ab}$ .
Step 3. Group  $a$  and  $b$  into a new cluster  $q$ .
        Set  $SIZE(q) = SIZE(a) + SIZE(b)$ .
Step 4. for every node  $c$  such that
         $E-R(c, a) = \text{true}$  or  $E-R(c, b) = \text{true}$ ,
        there are four possible cases:
            Case 1. if  $E-R(c, a) = \text{true}$  and  $E-R(c, b) = \text{true}$ , then
                    set  $E-R(c, q) = \text{true}$  .
                    assign to edge  $(c, q)$  a weight equal to  $(W_{ca} + W_{cb})$ .
            Case 2. if  $E-R(c, a) = \text{true}$  and  $E-R(c, b) = \text{false}$ , then
                    set  $E-R(c, q) = \text{true}$ .
                    assign to edge  $(c, q)$  a weight equal to  $W_{ca}$ .
            Case 3. if  $E-R(c, a) = \text{false}$  and  $E-R(c, b) = \text{true}$ , then
                    set  $E-R(c, q) = \text{true}$ .
                    assign to edge  $(c, q)$  a weight equal to  $W_{cb}$ .
            Case 4. if  $E-R(c, a) = \text{false}$  and  $E-R(c, b) = \text{false}$ , then
                    set  $E-R(c, q) = \text{false}$ .
Step 5. if (  $SIZE(a) = 1$  and  $r_a \geq 1$  ) then
        begin
        Step 5.1 Add edge  $(a, q)$  to the graph.
        Step 5.2 Assign  $+\infty$  to edge  $(a, q)$ .
        Step 5.3 Set  $r_a = r_a - 1$ .
        end
    else
        begin
        Step 5.4 Delete node  $a$ .
        Step 5.5 for every node  $c$  such that  $E-R(c, a) = \text{true}$ , do
                assign  $E-R(c, a) = \text{false}$ .
        end
Step 6. Repeat Step 6 for node  $b$ .
end
Step 7. for every node  $c$  such that (  $SIZE(c) = 1$  and  $r_c \geq 1$  ) do
    begin
    Step 7.1 Let  $r_c = k$ . Add  $k$  new nodes, called  $c$ , to the graph and
            for any one of these nodes duplicate the edges incident to
            the node  $c$  that has initially been in the graph.
    Step 7.2 Assign a  $+\infty$  weight to the edge connecting to any two of
            these new nodes to one another or connecting any one of
            them to the node  $c$  that has initially been in the graph.
    end

```

In Step 1, the value of function $SIZE$ at every node in the graph is set to one. Steps 2-6 are executed until there is no edge with a negative weight or there is only one node

in the graph. In Step 2, the edge with a minimum weight is chosen as a candidate for reducing communication cost. In Step 3, the two nodes of the edge chosen in Step 2 are clustered into a new node and the size of the new node is computed. In Step 4, the weights affected by the addition of the new node are modified. In Steps 5 and 6, the clustered nodes are deleted. When the node is a cluster or is not replicated, the node along with all the edges associated with it are deleted. When the node has replicates, we replace the node with its replicate and assign the largest possible weight, $+\infty$, to the edge connecting the new cluster and the node representing its replicate so that no object can be grouped with its replicate to the same cluster. In Step 7, the nodes that each represents only one object where the object has some replications to be considered are identified. If it turns out that k replications of the object are to be considered, then k new nodes of the object are created and the edges incident to any one of these new nodes has the same weight as that of the identical edge incident to the old node of that object. The edges that connect either any two of these new nodes to one another or any one of these new nodes to the old node of the same object, should be assigned a weight of $+\infty$. This measure is taken to insure that no object will be placed with its replication in one cluster.

The output is an undirected weighted graph in which every node models a cluster of object(s) and every edge has a positive weight which represents the degree of contribution to the overall performance of the system that can be made by the parallel execution of the two clusters represented by the two involving nodes. Note that a larger weight implies more gain can be obtained as a result of allocating the involving clusters on two different processors.

4.2 Time Complexity

The time complexity of the clustering algorithm described above is a function of m , and e' where m is the number of objects in the graph including replicated objects, and e' is the total number of edges if replicated objects were also included in the graph. Step 1 takes $O(n)$ time. Step 2 runs in $O(e')$. Step 3 has a constant running time. Each pass of Step 4 can run in $O(n)$ time. Because there are at most n objects involved, the time complexity of Step 5 is $O(n \times \log(e'))$. Step 6 is simply the repetition of Step 5. The while loop will be executed at most e' times. This makes the time complexity of the loop $O(\min(e', m) \times \max(e', n))$ which is equivalent to $O(e' \times m)$. Step 7 can also run in $O(e' \times m)$ in the worst case. Therefore, the time complexity of the clustering algorithm is $O(e' \times m)$ in the worst case. Clearly, if no replicated objects is considered, the worst-case time complexity of clustering algorithm will reduce to $O(e \times n)$.

Chapter 5

PROOF/L Front-end Translation

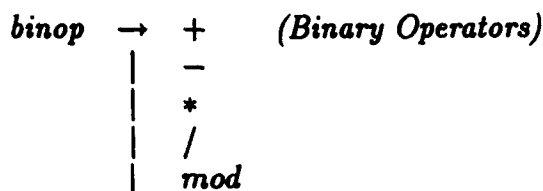
The PROOF/L front-end translator is the first stage in the process of transformation from PROOF/L to a target language. One of the major tasks performed by the front-end translator is to make the implicit parallelism present in PROOF/L code explicit. In this chapter, we will describe the translation rules which form the basis for the front-end translator. These rules cover a number of different functional forms and some special node insertions, such as the *copy* node. The UNIX tools: *lex* and *yacc* are used for the implementation of the front-end translator. An example program **Bounded Buffer** is used to illustrate the translation from PROOF/L to the IPR. We will also explain different kinds of detections and reconstructions for control constructs, such as **if** and **while**, and the method for generating the resultant IPR.

5.1 Syntax Rules of PROOF/L

In this section, we will present the concrete syntax rules of PROOF/L. The syntax rules include the functional forms in PROOF such as α (apply to all), β (distributed apply), γ (filter), **while** (loop), **if** (conditional) and **R** (pseudo function). The syntax also includes object-oriented features such as inheritance and body of active as well as pseudo active objects. Complete concrete syntax rules are listed as follows:

<i>proofl</i>	→	program name : <i>class_list obj_list body_list</i> end
<i>body_list</i>	→	<i>body_def</i> // <i>body_list</i>
		<i>body_def</i>
<i>body_def</i>	→	body of object name : <i>func</i>
<i>obj_list</i>	→	<i>obj_list obj_def</i>
		<i>obj_def</i>
<i>obj_def</i>	→	<i>active_opt</i> c_object <i>name_list</i> : instance of name <i>ins_opt</i>
<i>ins_opt</i>	→	(<i>name_list</i>)
<i>active_opt</i>	→	active
		pseduo active
<i>class_list</i>	→	<i>class_list class_def</i>
		<i>class_def</i>
<i>class_def</i>	→	class name <i>class_ins local_data_list super_class method_def</i> end class
<i>class_ins</i>	→	(<i>dcln_list</i>)
<i>super_class</i>	→	superclass : name (<i>dcln_list</i>) inherit : <i>inherit_opt</i>
<i>inherit_opt</i>	→	all
		<i>name_list</i>
<i>name_list</i>	→	name , <i>name_list</i>
		name
<i>local_data_list</i>	→	composition <i>local_data</i>
<i>local_data</i>	→	<i>dcln</i> X <i>local_data</i>
		<i>dcln</i>
<i>method_def</i>	→	<i>method method_def</i>
		<i>method</i>
<i>method</i>	→	method name (<i>method_io</i>) <i>guard_dcln</i> expression <i>func</i>
<i>guard_dcln</i>	→	guard (<i>bool_exp</i>)
<i>method_io</i>	→	<i>input_list</i> - > <i>output_list</i>
<i>input_list</i>	→	<i>dcln_list</i>
<i>dcln_list</i>	→	<i>dcln</i> , <i>dcln_list</i>
		<i>dcln</i>
<i>dcln</i>	→	name : <i>data_type</i>
		name : <i>class_name</i>
		name : <i>list_opt</i> (<i>data_type</i>)
		name : <i>list_opt</i> (name)
		name
		<i>data_type</i>
<i>data_type</i>	→	int
		boolean
<i>output</i>	→	<i>dcln</i>

<i>class_name</i>	→	name	
<i>list_opt</i>	→	list * list_opt	
		list	
<i>output_list</i>	→	output , output_list	
		output	
<i>inst_list</i>	→	inst , inst_list	
		inst	
<i>inst</i>	→	name = func	
<i>func</i>	→	λ (name) func	(lambda abstraction)
		; (func_list)	(sequential)
		let name = func in func	
		object name (inst_list)	(object instantiation)
		α name [func_list]	(alpha function form)
		β [func_list] [func_list]	(beta function form)
		γ [boolExp_list] [func_list]	(gamma function form)
		stmt	
<i>stmt</i>	→	while (boolExp , func) func	(while loop)
		if (boolExp , func , func) func	(conditionnl)
		R [name] func	(Pseudo function R)
		exp	
<i>exp</i>	→	aexp	
		exp aexp	
		binop aexp aexp	
<i>boolExp</i>	→	boolOp aexp aexp	
		not boolExp	
		true	
		false	
		λ (name) boolExp	
		(boolExp)	
<i>aexp</i>	→	name	
		integer	
		NIL	
		string	
		(func_list)	
		[func_list]	(List)
<i>boolOp</i>	→	=	(Boolean Operators)
		<>	
		<	
		<=	
		>	
		>=	
<i>func_list</i>	→	func , func_list	
		func	



5.2 Intermediate Program Representation

IPR is a directed graph $G = (V, E)$ in which V is a set of nodes and E is a set of directed edges. A node represents a computation or data object¹. An edge (u, v) represents dataflow from node u to node v . V can be divided into three types - *computation node*, *control construct node* and *list handling node*.

A *computation node* represents a function receiving input value(s) and generating output value(s). These functions are free from side effect, that is, they always produce the same result when the same input values are given.

Computation nodes include basic *numeric* and *boolean* operators, *constant*, *id* and *copy* nodes.

- Numeric and boolean operators includes operators such as $+$, $-$, $*$, $/$, $=$, $<$, $>$.
- A *constant* node represents a constant generator which produces the same specified value. There is no input to this node.
- An *id* node represents an identity function which always returns the same value as its input value.
- A *copy* node represents a duplicator, which receives an input and produces an appropriate number of copies having the same value as its input.

Control construct nodes are needed to specify the control flow among functions. Although the data flow dependency relationship is the dominating factor in dictating the execution flow of the program, in order to represent control functions, such as *if* and *while*, we need the *select*, *distributor* and *merge* nodes.

- A *select* node represents a conditional construction function. It receives input data i_1, i_2, \dots, i_n and control data c and returns an input i_i as an output according to the value of control data c .
- A *distributor* node represents a conditional construction function. It receives input data i and control data c and passes i to one of the output ports o_1, o_2, \dots, o_n according to the value of c .

¹Data object can also be considered as a special case of computation.

- A *merge* node represents a nondeterministic selector, which receives an arbitrary number of input data at a time and returns one, which arrives first to it. If more than one input arrives at the same time, they are chosen in an arbitrary order.

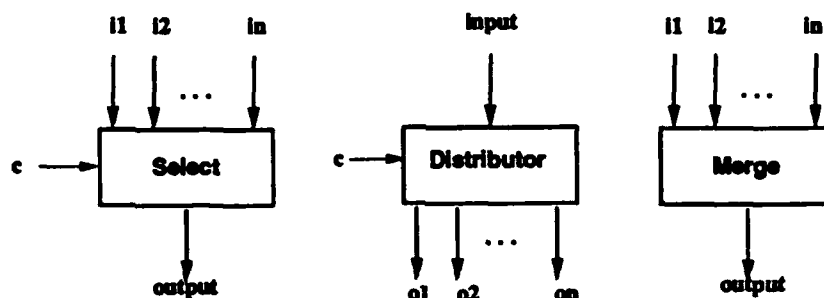


Figure 5.1: The IPR forms of select, distributor and merge node.

There are two kinds of *list handling nodes*: *construct* node and *split* node.

- *construct* node receives one or more input values and make them as a list.
- *split* node receives a list as an input and splits that list into values.

5.3 Translation Rules : From PROOF/L to IPR

In this section we will present the translation rules from PROOF/L to the IPR. Different syntax rules in Section 5.1 will be translated to different kinds of IPR form depending on their semantics. For example, the α functional form's semantic meaning is "apply to all". Therefore, we need to translate "apply to all" into a corresponding IPR to represent its semantics. In some cases, some special nodes need to be inserted, such as *latch*, *split* and *copy*. All these translation rules is presented in this section. For each translation rule, we will first present the functional form in PROOF/L and then the corresponding IPR form.

1. *Function Application* - apply function *func* to $e_1 e_2 \dots e_n$
 $\text{func } e_1 e_2 \dots e_n$ where $n \geq 1$

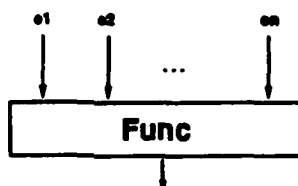


Figure 5.2: The corresponding IPR for *function application*.

2. *List* - a list consists of $e_1 e_2 \dots e_n$
 $[e_1, e_2, \dots, e_n]$ where $n \geq 1$

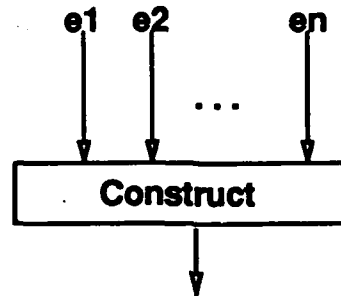


Figure 5.3: The corresponding IPR for *list*.

3. α Functional Form - apply to all
 $\alpha \text{ func } [e_1, e_2, \dots, e_n]$

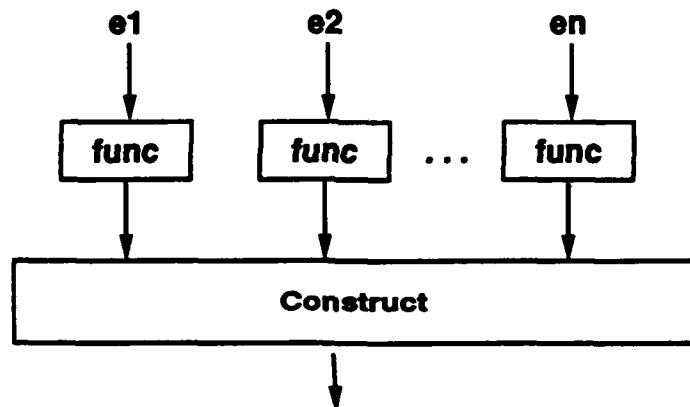


Figure 5.4: The corresponding IPR for α functional form.

4. β Functional Form - distributed apply
 $\beta [f_1, f_2, \dots, f_n] [e_1, e_2, \dots, e_n]$

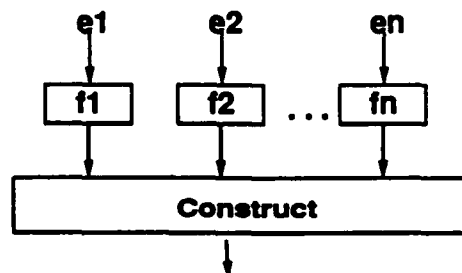


Figure 5.5: The corresponding IPR for β functional form.

5. γ Functional Form - filter

$\gamma [b_1, b_2, \dots, b_n] [e_1, e_2, \dots, e_n]$

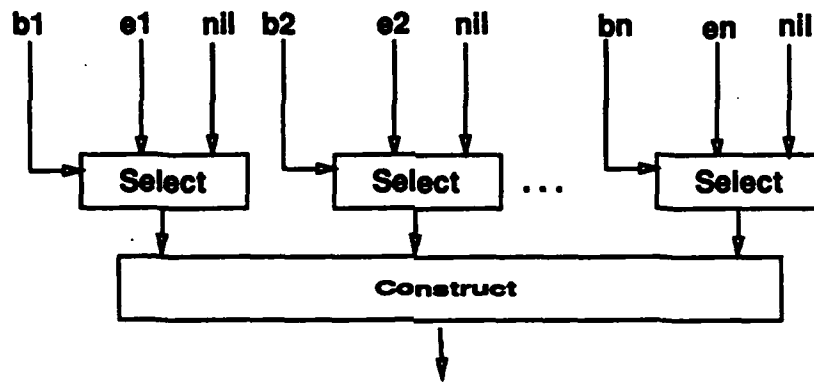


Figure 5.6: The corresponding IPR for γ functional form.

6. *Lambda* Abstraction

$\lambda x.exp$

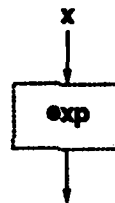


Figure 5.7: The corresponding IPR for *lambda* abstraction.

The dashed box will be replaced by the actual definition of *exp*.

7. *While* Functional Form - while *b* is true, continue applying *e* to *x*

while(b,e) x

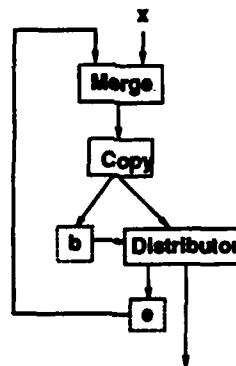


Figure 5.8: The corresponding IPR for *while* functional form.

The dashed boxes will be replaced by the actual definitions of *b* and *e*.

8. *If Functional Form* - if b is true, then apply e_{then} to e ; otherwise, apply e_{else} to e .
 $if(b, e_{then}, e_{else}) e$

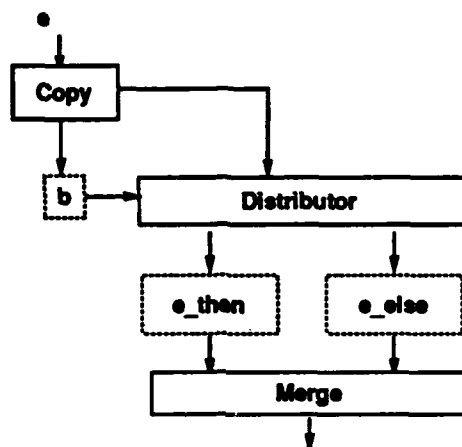


Figure 5.9: The corresponding IPR for *if* functional form.

The dashed boxes will be replaced by the actual definitions of b , e_{then} and e_{else} .

9. *Split* Node Insertion

```

class obj_class
  composition  $c_1 \times c_2 \times \dots \times c_n$ 
  :
  method fooobj : obj_class
  :
end class
  
```

Although objects can be parameters of a method, the operations of a method of an object mostly apply to the local data of the object. Hence, in this case we first split the object into its local data. The IPR form is :

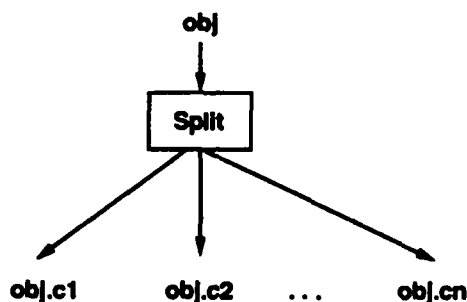


Figure 5.10: The corresponding IPR for *split* node insertion.

10. Copy Node Insertion

Some different functions may use the same input as shown in the following figure:

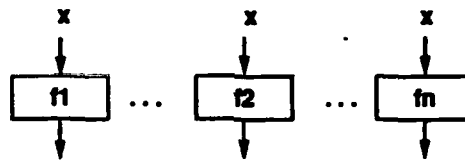


Figure 5.11: Functions f_1, f_2, \dots, f_n have the same input x .

In this case, a *copy* node is inserted into the IPR form.

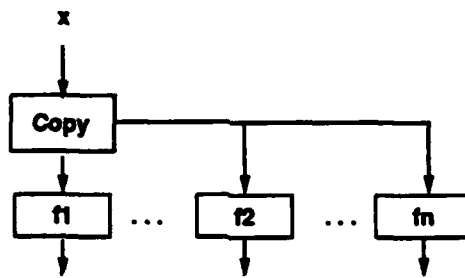


Figure 5.12: The resulting IPR after a *copy* node is inserted.

11. Sequential Function Form - executes f_1, f_2, \dots, f_n sequentially.

$;(f_1, f_2, \dots, f_n)$ where $n \geq 1$

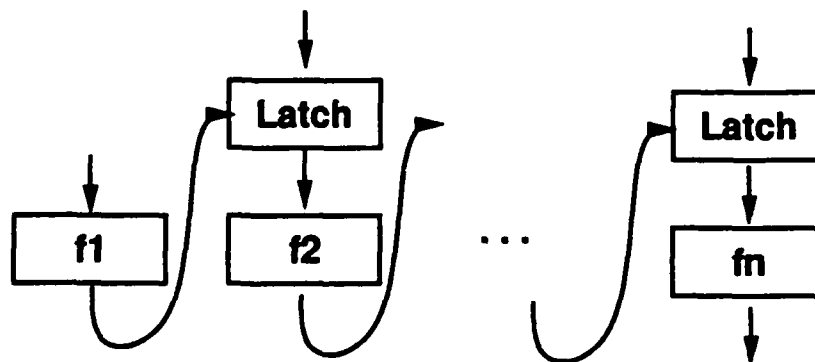


Figure 5.13: The corresponding IPR for *sequential* functional form.

The latch node is a special node just for control flow usage. This node has two inputs, control and data-in. The data from data-in cannot go through the node unless the control line is fired.

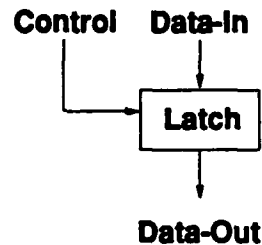


Figure 5.14: The input/output of a *latch* node.

5.4 Textual Form of IPR

IPR is a kind of graphical form and a typical node in IPR is shown in Figure 5.14. We need a textual form in order to save IPR into files. In this section, we introduce the textual form of IPR.

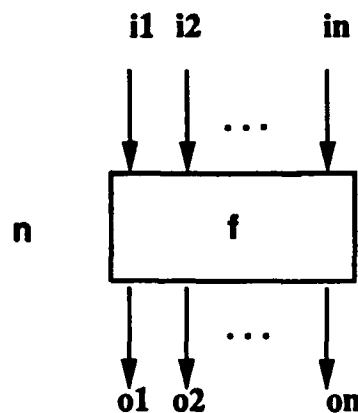


Figure 5.15: A typical IPR node.

where, n is the number of this node, f is the function of this node, such as constructor, selector ..., i_k , $k = 1, 2, \dots, n$ are the input nodes, and o_k , $k = 1, 2, \dots, n$ are the output nodes.

This node can also be represented in the following tabular form:

Node	Function	Inputs	Outputs
n	f	i_1, i_2, \dots, i_n	o_1, o_2, \dots, o_n

Now we would like to present a bounded buffer as an example to illustrate our process of translation. First, we will present the PROOF/L source code for bounded buffer and then show how to apply the translation rules introduced in Section 5.3. Finally, the resultant IPR output is presented.

The PROOF/L code of the example Class Bounded_buffer is given as follows:

```
class BoundedBuffer(itentype, size)
  composition store:list(itentype) × count:int
  method put buf x
    guard(< buf.count size)
    expression
       $\beta$ [(append-right x), inc] [buf.store,buf.count]
  method get buf
    guard(> buf.count 0)
    expression
      [  $\beta$ [tail,dec] [buf.store,buf.count], head(buf.store) ]
end class
```

This is a complete definition of a class in PROOF/L. Now, we would like to present the corresponding IPR of each method.

- Method get

```
method get buf
  guard(> buf.count 0)
  expression
    [  $\beta$ [tail,dec] [buf.store,buf.count], head(buf.store) ]
```

The parameter is an object, and according to the translation rule (9) of Section 5.3 a split node is inserted to split the object into its local data, store and count. The output of method get is a list, and hence rule (2) is applied. The first element of this list is a β functional form and hence rule (4) is applied. The second element is an ordinary function application and rule (1) is used. We can see both tail and head use the buf.store as input. According to rule (10), a copy node is inserted. The resultant IPR form of method get is in Figure 5.16. The corresponding textual form is given as follows:

Node	Function	Inputs	Outputs
1	CONSTRUCT	2,6	OUTPUT
2	CONSTRUCT	3,4	1
3	tail	5	2
4	dec	7	2
5	COPY	7	3,7
6	SPLIT	8	5,4
7	head	5	1
8	buf	INPUT	8

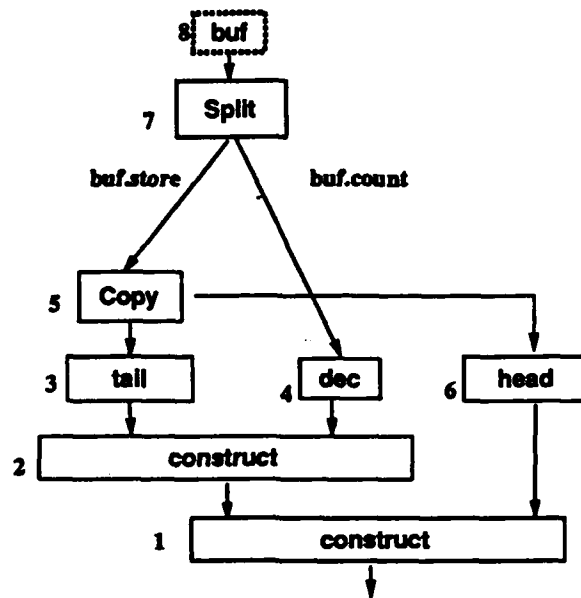


Figure 5.16: IPR for method *get*.

- Method *put*

```

method put buf x
  guard(< buf.count size)
  expression
     $\beta$ [(append_right x), inc] [buf.store, buf.count]

```

There are two parameters of method *put*. The object parameter *buf* should be first split into its local data. The body of method *put* is mainly a β functional form; therefore, apply rule (4). The application of *append_right* and *inc* is translated by rule (1).

The resultant IPR is in Figure 5.17 and the corresponding textual form is shown as follows :

Node	Function	Inputs	Outputs
1	CONSTRUCT	2,4	OUTPUT
2	append_right	3,5	1
3	x	INPUT	2
4	inc	5	1
5	SPLIT	5	2,4
6	buf	INPUT	5

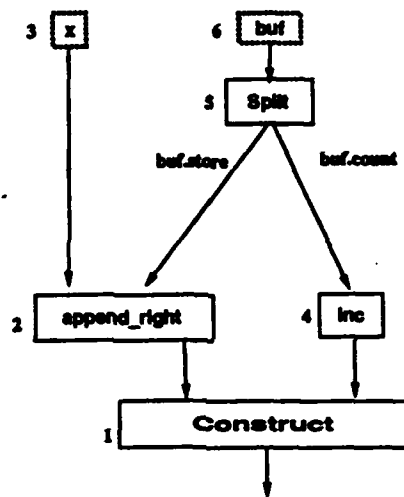


Figure 5.17: IPR for method *put* .

5.5 Implementation of PROOF/L Front-End Translator

In this section, the implementation of the PROOF/L front-end translator will be presented. The following figure is the architecture of the front-end translator.

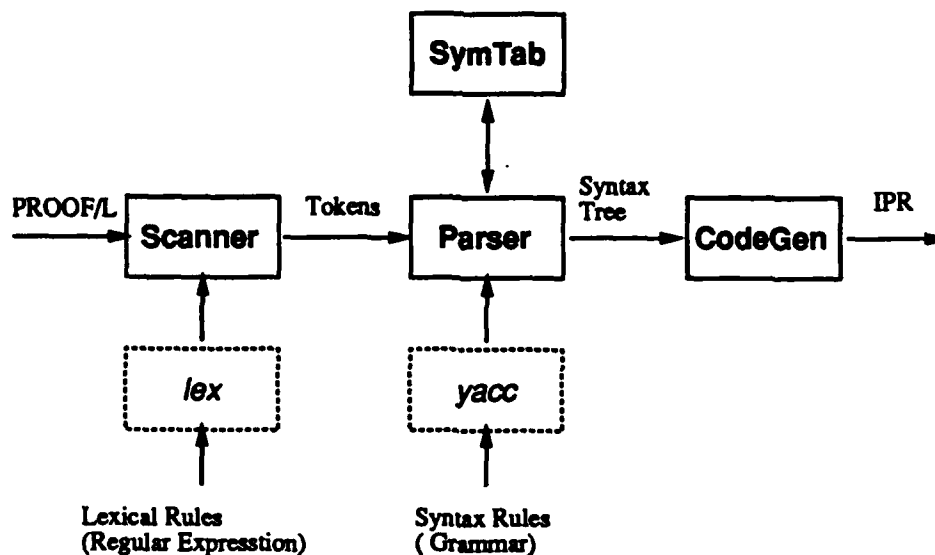


Figure 5.18: Architecture of the PROOF/L Front-End Translator.

There are four main modules: scanner, parser, code generation and symbol table handler. The scanner and parser are coded with the aid of UNIX language tools *lex* and *yacc*. We will present each of these four modules separately.

5.5.1 Lexical Analyzer

The function of a lexical analyzer is to group the input character stream into a token stream and as input of the latter parser phase. A token is a basic element of parsing.

In this part, we use the language tool *lex* generating the code of lexical analyzer. We can input the lexical rules(in regular expressions) to *lex* and it will generate the corresponding finite state machines for lexical analysis.

The input format of *lex* is divided into three parts:

<definitions>

%%

<rules>

%%

<programmer subroutines>

The first part *<definitions>* and third part *<programmer subroutines>* are optional.

In the first part *definitions*, we can specify some sets of the lexical rules in the next part. For example,

letter	[a-zA-Z]
digit	[0-9]
letter_or_digit	[a-zA-Z_0-9]
sign	[+-]

In the second part *<rules>*, we can use these defined sets to express the lexical rules. For example, the lexical rules for integer numbers:

```
digit+ {
    yylval.y_int = atoi(yytext);
    return token(INTEGER);
}
```

The left-hand side part is the regular expression of an integer and the right-hand side part is the corresponding actions of an integer token: converts the text into the number and return a token INTEGER.

The last part *<programmer subroutines>* consists of some C routines written by users.

Another part of this module is the screener. The function of screener is to distinguish key words from identifier because both are the same in structure and cannot distinguish by regular expressions. Here we only maintain a sorted key word table and use binary search to accomplish this.

5.5.2 Parser

The function of a parser is to check the correctness of input and then generate the corresponding abstract syntax trees. Here we use another language tool *yacc* to generate the code for parsing.

The input format of *yacc* is similar to that of *lex*. It also consists of the same three parts (definitions, rules and programmer subroutines) and they are also separated by two "%%". Only the second part is compulsory and the other two are optional.

The first part is the definitions. We need to give the definitions of tokens, return types, priority between operators and start rule of the grammar. For example,

```
%token          INTEGER
...
%type    <y_int>  INTEGER
...
%start   proofl   --
```

The second part is the most important part, including grammar rules and corresponding actions. For example,

```
proofl  :  PROGRAM ID COLON class_list obj_list body_list END
{
        body();
}
;
```

This is the main grammar rule of a PROOF/L program. It begins with the reserved word **program** and the name of this program. After a ";", the rest of the program is the class declarations and object declarations. Finally, it is the list of bodies of active objects and ended with the reserved word **end**. Similar to *lex*, between a pair of "{" and "}" is the corresponding action part of this rule. For the example above, the action is calling the function `body()` to build the object body list.

The grammar rules part has been presented in this section. The corresponding actions mostly are the code generation. Code generation will generate different IPR according to different syntax.

The rest of the actions are symbol table manipulations. They include the operations

of method table, class table and object table. These will be discussed later.

5.5.3 Code Generation

The code generation part is mainly an implementation of those translation rules. It can be summarized in the following procedure:

1. Recognize the syntax structure and select the corresponding translation rule. This is actually the part of parser and it is recursively executed.
2. According to the selected translation rule, generate the component IPR nodes. Label every node with a unique sequence number. This number is used to build the input/output relation among other nodes.
3. Build the input/output relations among nodes. Here every IPR node is attached with two arrays for its input and output nodes respectively. If its input or output are from outside, i.e., parameters of method or it is a constant node, a -1 is assigned to be its input or output.
4. Analyzes the connected nodes and adds SPLIT and COPY node(s) if they are necessary. A COPY node is added when there is a common input of several nodes. For example, the parameters of method and bound variables of λ functional form are all common inputs. A SPLIT node is added when an object is a parameter of its own method and an object is used in its own body. Because an object is implemented as a list, the list must be split first in order to use its local data.
5. Perform the data dependency analysis for control constructs. IPR is basically an intermediate form in data dependency. However, for those control constructs, such as while and if, they may or may not have data dependency between nodes. Hence, we should first find out whether data dependency exists. This can be done by searching the bounded variable in the then-part and else-part of if functional form or the loop body of the while functional form. If bounded variable cannot be found, this is pure control dependency. Similar to the sequential functional form, we can use LATCH node to enforce control dependency. Therefore, for those pure control dependency in these control constructs, LATCH nodes are added to its inputs to enforce control flow.

5.5.4 Symbol Table Handling

Because PROOF/L is an object-oriented applicative language, the only symbols we need to deal with are objects. To maintain the symbol table of objects, first a class table should be maintained and every entry in the class has two lists to point to its method list and local data list respectively. When an object is declared, an entry is added to the object table and this entry has a pointer to its class in the class table. We can use this to check if every declared object belongs to a declared class, and whether

local data access and method invocations are legal. For those active or pseudo active objects, their entries in the object table also have pointers to the body list.

The relations among the tables are shown in Figure 5.19.

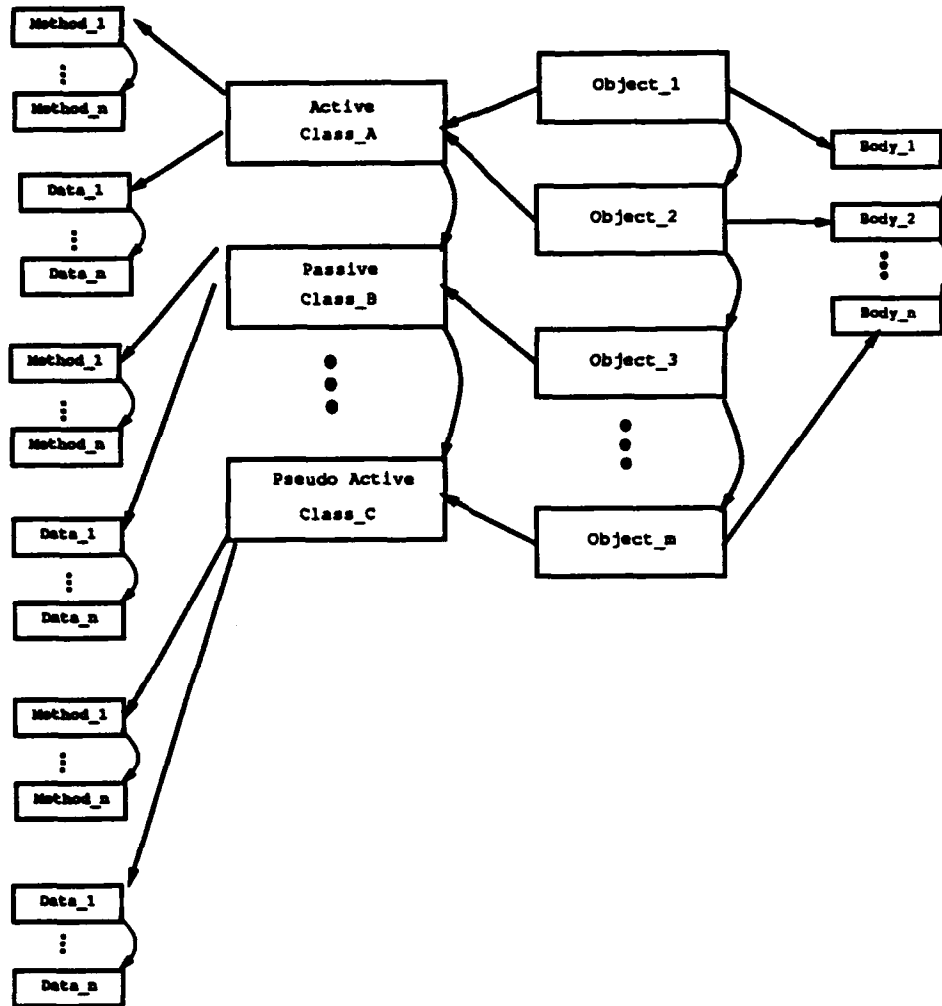


Figure 5.19: The structure of the symbol table.

5.6 Limitations

The current implementation has the following limitations:

- Built-in data types: Now only integer is supported. In the future, for better programming support, more built-in data types should be supported such as floating point, character and string.
- Abstract data types: In the current version, only list is supported. To a wider range of applications, more abstract data types should be supported. For ex-

ample, for supporting scientific computing such as matrix and vector processing, arrays should be supported.

- **Type Checking:** Because currently only integer is supported, a very simple type checking is implemented in the parser.
- **I/O facilities:** Now I/O facilities is not supported in the language. Because PROOF/L is an object-oriented language, in order to add I/O facilities, we can implement some I/O objects, like *cin* and *cout* for standard I/O in C++.
- **Inheritance:** Although the inheritance syntax is supported in the parser, the part of the symbol table has not yet been fully implemented. However, this can be accomplished by adding some hierarchical structures and duplicate mechanism.

Chapter 6

Grain Size Analysis

The goal of grain size analysis is to reduce the completion time of the program by minimizing the communication time without sacrificing parallelism. When we discussed partitioning in Chapter 4, we focused on exploring parallelism among the objects. When we consider grain size determination, we will find proper grain sizes within each object. In other words, we consider each object as an independent program throughout this chapter¹.

In this chapter, we will present three grain size determination algorithms for three patterns of parallelism: tree parallelism, graph parallelism and pipelined parallelism. For tree parallelism, we will present an efficient heuristic grain size determination algorithm and show that this algorithm can find optimal grain sizes in certain cases. We will then generalize this algorithm for the case of graph parallelism. In both cases, we compare the results of our approach with the existing grain size determination approaches. We will also present a grain size determination algorithm in case of pipelined parallelism. Then, we will describe a method to modify the IPR form to incorporate the information we have obtained from grain size analysis and partitioning.

In most of the approaches to partitioning and allocation of the tasks, or the schedulable units in an object, information regarding the execution times of the tasks and the communication times between the tasks is assumed to be available as the input, and thus the problem of obtaining such information has not been addressed [37, 38, 44]. However, in order to perform the grain size analysis based on the tradeoff between parallel execution and communication overhead, we need to estimate the execution time of each node in IPR and the communication time between two adjacent nodes. In our approach, we will obtain the information on the execution and the communication time by estimating the execution time for the simple nodes defined in IPR and the communication time by examining the type of information of the data being transmitted. The estimation can be done statically by analyzing the assembly language code generated for these simple nodes.

¹In fact, our grain size determination approaches can also be used for object level parallelism, if we have the information regarding the execution time of the tasks and communication time between the tasks.

6.1 Grain Size Determination

In this section, we will discuss the existing grain size determination strategies and the assumptions we will have for our grain size analysis.

The existing grain size determination strategies can be divided into two categories based on how the grain size is determined: programmer control and automatic determination. In programmer-controlled approaches, the programmer is fully responsible for determining the grain sizes as well as explicitly expressing the parallelism. The programmer can use parallel language constructs indicating how the tasks are executed in parallel. When a programmer has specific information about the behavior of a program, he can determine the sizes of tasks. When that program is ported to a different parallel processing system, the sizes of tasks need to be changed to better fit the new processors. In addition, it may not be easy for the programmer to make decisions on the sizes of the tasks due to lack of information.

On the other hand, in the category of automatic-determination approaches [48, 52, 53, 54], grain sizes are determined automatically. This category of approaches can be further divided into two classes: compiler approach and run-time approach. In a compiler approach, the programmer does not provide any information regarding the granularity. During compilation, heuristics are used to statically determine the sizes of tasks [48, 53, 54]. One disadvantage of this kind of approaches is that some information may not be available before the run-time.

In a run-time approach, only simple heuristics can be applied to determine the sizes due to costly overhead. For example, as in [52], each recursive function call creates a new task to be assigned to a processor. In this case, since functional programming involves frequent recursive function calls, it is likely that too many small tasks will saturate the system. In general, such a run-time approach ignores the size of tasks under the assumption that there are reasonably many processors available. The automatic determination approach looks more promising since the programmer does not need to worry about the grain size at all.

In our approach, the grain size is determined based on the analysis of the execution and communication time which can be obtained during the compilation time.

We make the following assumptions about the underlying parallel processing systems:

- The system is an MIMD machine consisting of fully connected identical processors having the same processing capability.
- Each processor has a capability of performing program execution and I/O simultaneously.
- The communication cost between two processors depends only on the data size to be transmitted. Currently, we ignore the time required to set up the communication.

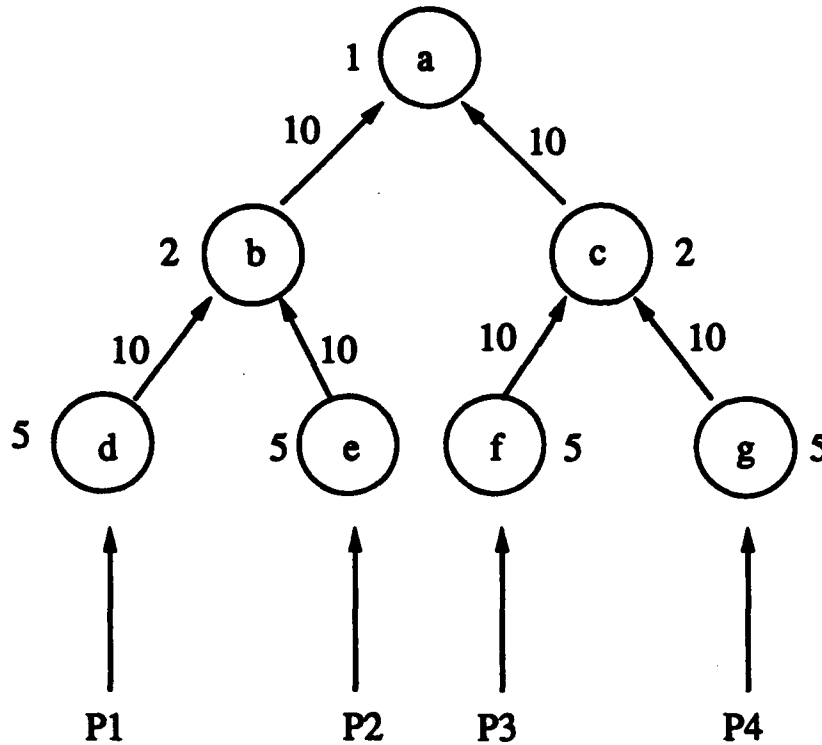


Figure 6.1: An IPR representation for Φ_1 .

- Communication cost between two tasks residing on the same processor is negligible and hence is not considered.

The following notations will be used in this section:

Definition 6.1.1 The *execution time* for a node p , denoted by $e(p)$, is the amount of time required to complete the task represented by p without being interrupted.

Definition 6.1.2 The *communication time* between two nodes p and q , denoted by $c(p, q)$ is the amount of time required to transmit data from p to q under the assumption that p and q are assigned to adjacent processors.

Definition 6.1.3 The *completion time* for a program Φ with k processors, denoted by $D(\Phi, k)$, is the amount of time required to finish all the tasks and communication in the program Φ with the k processors.

Suppose that we have a PROOF/L program, called Φ_1 , such as $a(b(d(v_1), e(v_2)), c(f(v_3), g(v_4)))$. The IPR for Φ_1 is shown in Figure 6.1. The graph consists of seven tasks, a, b, c, d, e, f, g , and six edges representing data precedence relations among the tasks. This tree-type parallelism is a typical form resulting from a divide-and-conquer algorithm.

Assuming that there are enough processors, one for each task. A simple approach to assign the tasks to the processors would be to assign four tasks d, e, f and g to four different processors P_1, P_2, P_3 and P_4 , respectively. After completing the execution of these four tasks, P_1 and P_3 continue the execution of the tasks b and c, respectively. Then, P_1 executes a to complete the execution. In this case, $D(\Phi_1, 4)$ can be calculated as follows:

$$D(\Phi_1, 4) = 5 + 10 + 2 + 10 + 1 = 28$$

This result is not desirable since if we only utilize one processor,

$$\begin{aligned} D(\Phi_1, 1) &= \sum_{x=a}^g e(x) \\ &= 5 \times 4 + 2 \times 2 + 1 \\ &= 25. \end{aligned}$$

Thus, $D(\Phi_1, 1) < D(\Phi_1, 4)$ and there is no gain in parallel processing because the communication overhead has overshadowed the gain of parallel execution.

6.2 Tree Parallelism

We will first define the terms to be used for tree parallelism.

Definition 6.2.1 A *one-level subtree* is a subset of nodes v_0, v_1, \dots, v_n in a tree such that v_0 is a parent node of all the nodes v_1, v_2, \dots, v_n .

In the following, we call a one-level subtree simply a subtree. The number of subtrees in a tree is the same as the number of non-leaf nodes.

Definition 6.2.2 A *task precedence tree* T_p is a tree in which each node represents a task and each edge specifies the data dependency relation between two nodes.

Parallelism obtained from the divide-conquer strategy can lead to the parallelism of a tree pattern and thus be represented by a task precedence tree T_p .

Definition 6.2.3 A *gain tree* T_g of T_p is a weighted tree in which each node, called a *gain node*, represents a subtree in T_p and each edge represents data dependency relations among the nodes. Each gain node has a weight, called *gain*, corresponding to the amount of maximum contribution to reducing the completion time when the nodes in the corresponding subtree are grouped into a node.

A gain for a subtree consisting of n_1, \dots, n_m is denoted by $GAIN(n_1, \dots, n_m)$. Our grain size determination approach can be considered as a *horizontal* grouping or partitioning process in that a set of adjacent nodes, i.e., a subtree is considered as a

candidate for grouping. The essential part of our grain size determination approach is to estimate the possible contributions which can be made by grouping the adjacent nodes. Our approach consists of two parts: build a gain tree from a given input task precedence graph, and determine grain sizes from the gain tree. The gain tree can be built by analyzing each subtree in the task precedence tree using the following procedure:

The input to the procedure is a subtree, consisting of a node s and its child nodes n_1, \dots, n_m , and its output is $GAIN(s, n_1, \dots, n_m)$.

Procedure Gain Analysis

Step 1 Calculate the total execution time for all nodes in one processor.

Let τ be the total execution time. Then,

$$\tau = e(s) + \sum_{i=1}^m e(n_i).$$

Step 2 Let $\sigma_i = e(n_i) + c(n_i, s)$, $i = 1, 2, \dots, m$.

Find a node n_k , $1 \leq k \leq m$, such that σ_k is the second largest among σ_i , $i = 1, 2, \dots, m$.

Step 3 If $\sigma_k + e(s) > \tau$, then

$$GAIN(s, n_1, n_2, \dots, n_m) = \sigma_k + e(s) - \tau$$

else

$$GAIN(s, n_1, n_2, \dots, n_m) = 0.$$

In Step 1, the total execution time τ is determined by summing all the execution time of the nodes in the subtree. In Step 2, calculate the time required to complete the computation represented by the subtree where the nodes in that subtree are not grouped together. Since one of the child nodes and the node s can be scheduled to the same processor and the scheduler can choose a node n_j , $1 \leq j \leq m$, such that σ_j is the largest, the second largest schedule length σ_k is calculated and used as the actual time required to complete the task represented by s, n_1, \dots, n_m . In Step 3, a gain is calculated by comparing the total execution time τ with the actual completion time σ_k calculated in Step 2. If there is a positive gain, then the amount of the gain calculated is assigned to the gain node. Otherwise, the gain is set to zero.

Step 1 requires $O(1)$ time, Step 2 requires $O(m)$ time, where m is the number of child nodes and Step 3 requires $O(1)$ time. Thus, the time complexity of the procedure Gain-Analysis is $O(m)$.

Now, we build a *gain tree* T_g from a task precedence tree T_p using the procedure Gain-Analysis in the following manner:

The input to this procedure is a task precedence tree and its output is a gain tree.

Algorithm 6.2.1 Build Gain Tree

For all subtrees t , do

Let t_s consist of a node s and its child nodes n_1, n_2, \dots, n_m .

Step 1 Call Gain-Analysis (s, n_1, n_2, \dots, n_m).

Step 2 Connect the gain node to the existing gain nodes.

In Step 1, Gain-Analysis is called for each subtree to determine the possible contribution to the reduction of the completion time when all the nodes in the subtree are grouped. In Step 2, the newly-created gain node for the current subtree is connected to the existing gain nodes. The construction of the gain tree can be done by omitting leaf nodes from the original task precedence tree and associating the gain calculated in Step 1 with the parent of the corresponding subtree. Step 1 requires $O(m)$, where m is the number of the child nodes. Since Step 1 is executed for each subtree and the number of subtrees is bounded by the number of non-leaf nodes in the tree, the algorithm needs to visit each node once. In Step 2, each node also needs to be visited once. Thus, the time complexity of Algorithm 6.2.1 is $O(n)$, where n is the number of nodes in the tree.

Once the gain tree is built, the grain size can be determined by selecting groups heuristically. Our grain size determination is based on the observation that the contribution from the nodes close to the root node² propagates to the other nodes. In order to illustrate this, suppose that we have simple gain trees as shown in Figure 6.2 in which $v_i, i = 1, 2, 3$, represent a set of gain nodes and x, y, z represent an amount of the gain for each node, respectively. In Figure 6.2 (a), the precedence relations are $v_2 \rightarrow v_1$ and $v_3 \rightarrow v_1$. In Figure 6.2 (b), the precedence relations are $v_1 \rightarrow v_2$ and $v_1 \rightarrow v_3$. The goal of the gain analysis is to select the two gain nodes for grouping in order to increase the overall contribution to the reduction of the completion time. Note that each gain node in the gain tree represents a contribution when a set of nodes in the subtree of the task precedence tree is grouped together. Thus, we mean that grouping of a gain node is to group together a set of nodes in the task precedence tree. The overall contribution is determined as follows:

$$\begin{aligned} &\text{overall contribution} \\ &= x + \max(y, z), \text{ if } \min(y, z) \leq x \\ &= \min(y, z), \text{ if } \min(y, z) > x \end{aligned}$$

Thus, in case of $\min(y, z) \leq x$, if $y > z$, v_1 and v_2 are grouped together, otherwise v_1 and v_3 are grouped together. In case of $\min(y, z) > x$, v_2 and v_3 are grouped together.

Note that the rule for determining the overall contribution presented above can be applied to both gain trees shown in Figure 6.2. It implies that our gain size analysis technique can be used for the analysis of both in-tree form as in Figure 6.2 (a) and out-tree form as in Figure 6.2 (b). The following is an algorithm to determine grain sizes for tree parallelism:

The input to this algorithm is a gain tree T_g consisting of a set of nodes n_1, n_2, \dots, n_m ,

²The root node in a tree is a node having depth of 0.

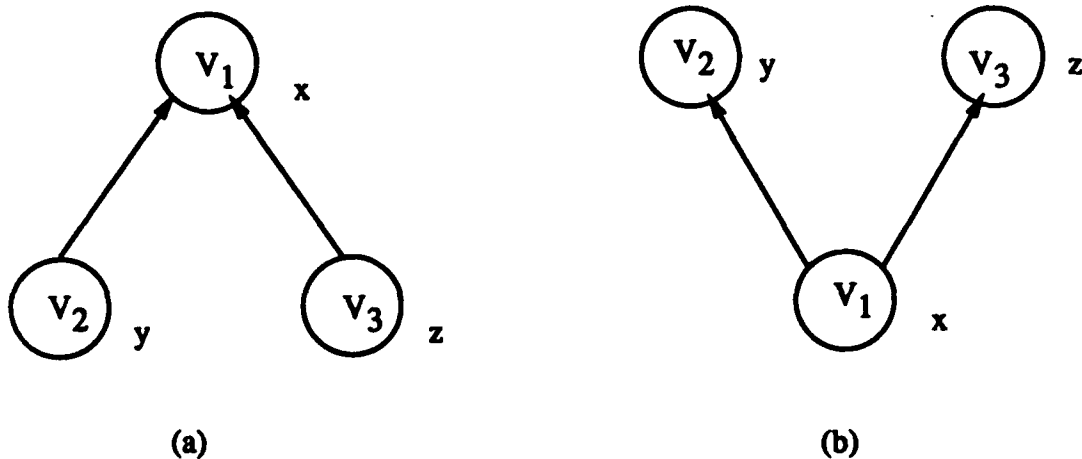


Figure 6.2: Two simple gain tree examples: (a) in-tree form, and (b) out-tree form

and its output is a set of grains.

Algorithm 6.2.2 Determine Grain Size

- Step 1** Initialize all the gain nodes n_i , $i = 1, 2, \dots, m$, as 'ungrouped'.
- Step 2** Sort n_i , $i = 1, 2, \dots, m$, in T_g using the gain as a primary key in descending order and the depth of each gain node as a secondary key in ascending order.
- Step 3** Get the gain node n_l whose gain is the largest in the sorted list.
 - Step 3.1** If n_l is not a root node, and both the parent of n_l and all the sibling of n_l are set 'grouped', then go to Step 3.4.
 - Step 3.2** If two or more child nodes of n_l are set 'grouped', then go to Step 3.4.
 - Step 3.3** Set n_l as 'grouped'.
 - Step 3.4** Delete n_l from the list.
 - Step 3.5** If there is a gain node with a positive gain, then go to Step 3, else Stop.

Algorithm 6.2.2 determines which subtrees need to be grouped by analyzing gains of the possible candidates. Step 1 requires $O(m)$ time to visit each gain node once, where m is the number of the gain nodes. In Step 2, $O(m \log m)$ time is required to sort m nodes. Step 3 must visit all the adjacent nodes of n_l at most once for each gain node. Because the number of the adjacent gain nodes is the same as the number of edges in n_l and each edge will be visited at most twice, the overall time complexity in Step 3 is bound to $O(e)$ in which e is the number of the edges in T_g . Therefore the

time complexity of this algorithm is $O(\max(m \log m, e))$. Note that in case of trees, the time complexity is $O(m \log m)$ since $m \log m$ is always greater than e .

In remainder of this section, we compare our approach to McCreary's approach [53] to demonstrate the efficiency of our approach. McCreary uses an algorithm of the time complexity $O(n^3)$, where n is the number of the nodes in the task precedence tree, that decomposes a graph into a set of *clans* that are classified as *primitive*, *linear*, or *independent*. When the clans are labeled as independent, the possibility of parallelization exists. However, when the clans are labeled as primitive or linear, they are grouped together and executed sequentially. In order to compare our approach to McCreary's, we use the same example used in [53], whose task precedence tree is shown in Figure 6.3. Every node has a unique number within the circle representing that node, and a weight is attached to the node. The communication cost is assigned to each edge. For instance, a node 9 has a weight 1 and each of the three edges has the communication cost 18. Using McCreary's approach, the schedule and its completion time are shown in Figure 6.4.

The problem with McCreary's approach is that it begins to search for the candidates for grouping from the bottom of the tree. Once the grouping is done at the lower part of the tree, the grouping at the higher level is less likely to occur since such grouping may have to sacrifice the possibility of parallel execution at the lower level. In addition, the key concept of the graph decomposition approach, *clan*, is determined without using information regarding the execution time of the nodes and communication time between the nodes.

Our grain size determination approach uses such information to determine the proper grains at the beginning. By analyzing the gain locally in each subtree, we build the gain tree. Then, we first select the largest gain node in the gain tree as the candidate for the grouping and continue to select the next largest gain node until all the gain nodes with positive gain are processed. The gain tree for this example is shown in Figure 6.5, where a calculated gain is shown within each gain node and the nodes in the task precedence tree shown in Figure 6.3 forming a gain node are identified besides the gain node.

From the information obtained in the gain tree, we can determine five grains as follows:

$$\begin{aligned} C1 &= \{ 1, 2, 9, 10, 13, 14, 15 \} \\ C2 &= \{ 5, 6, 11 \} \\ C3 &= \{ 7, 8, 12 \} \\ C4 &= \{ 3 \} \\ C5 &= \{ 4 \} \end{aligned}$$

Using these grains, we can show the two schedules and their completion time in Figure 6.6, one for four processors another schedule for five processors.

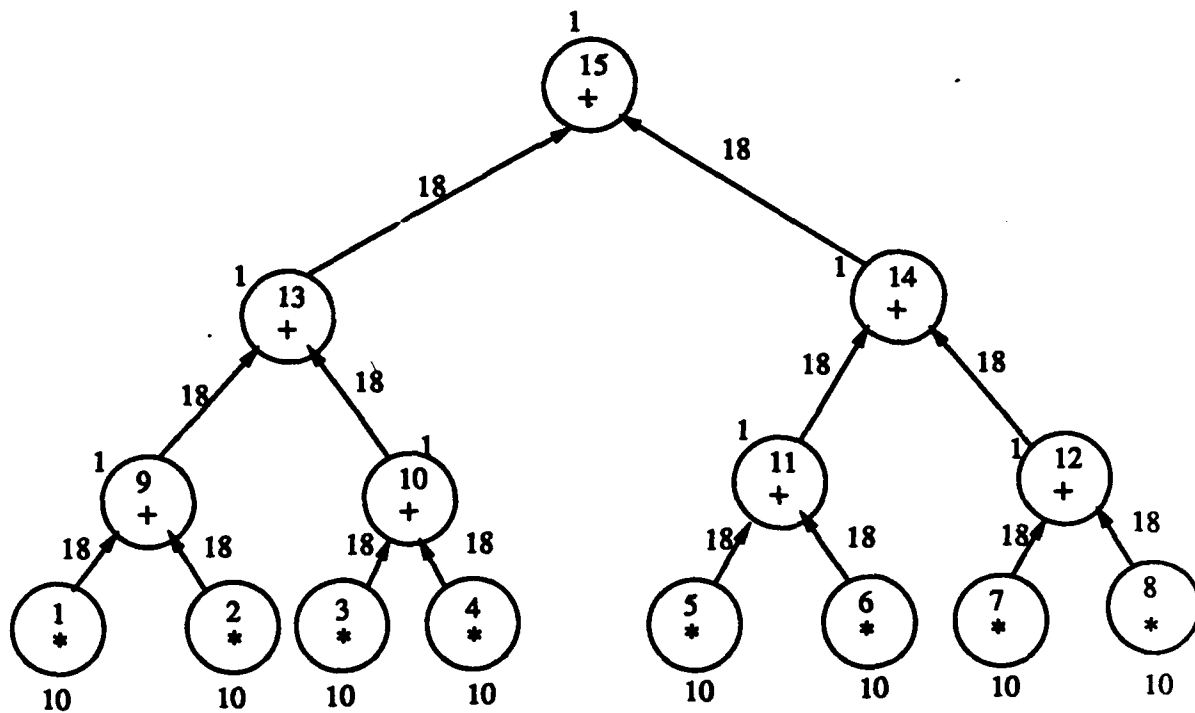


Figure 6.3: The task precedence tree of the tree parallelism example.

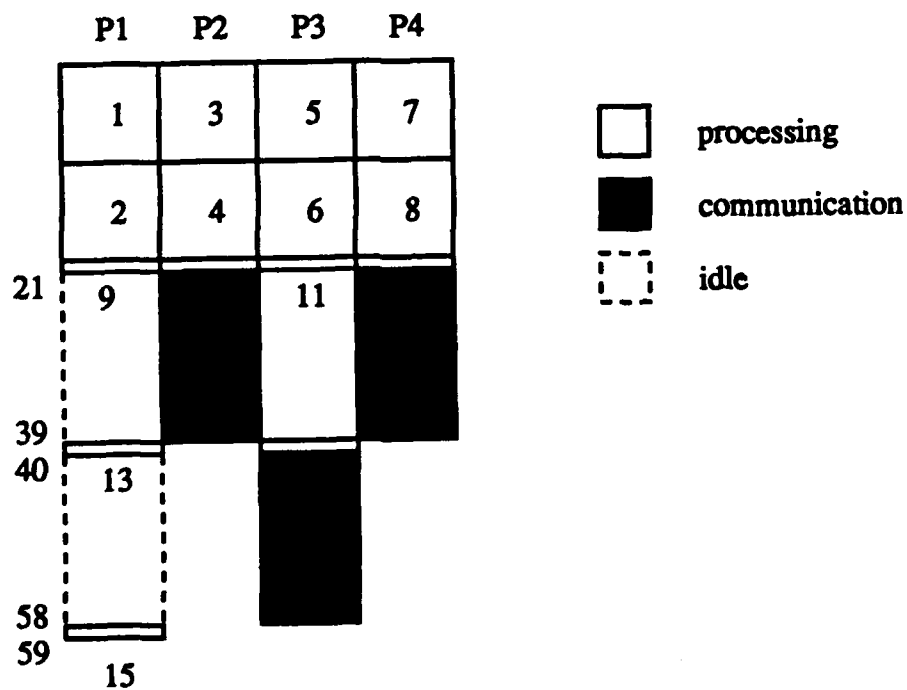


Figure 6.4: The schedule obtained from McCreary approach.

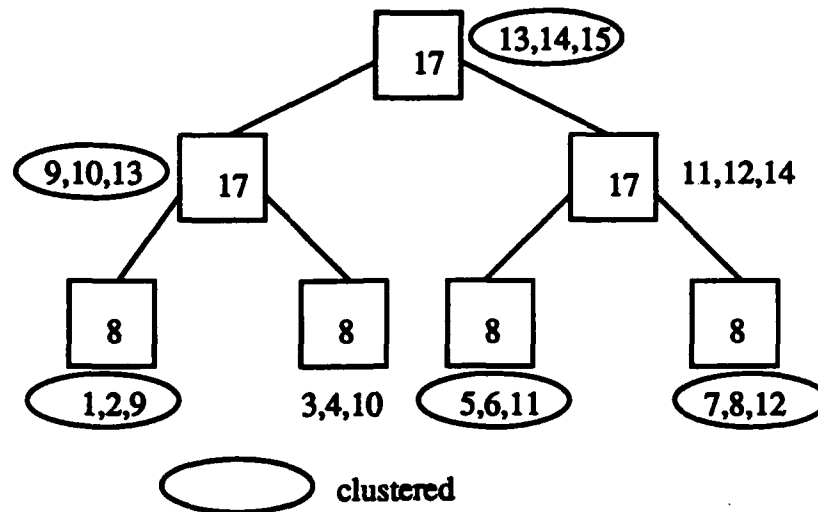


Figure 6.5: A gain tree for the tree parallelism example.

In the following, we will present a task allocation strategy which can produce an optimal solution when the following conditions are met:

- The grain size analysis indicates that no more grouping of grains is needed to reduce communication time among the tasks.
- The task precedence graph is a tree.
- There are a sufficient number of processors in the system so that the leaf nodes in the task precedence graph can be executed simultaneously, if necessary.

This algorithm can be considered as a variant of list scheduling, and uses the concept of the gain analysis. The input to this algorithm is a task precedence tree, and its output is a set of grains.

Algorithm 6.2.3 Task allocation for tree

- Step 1** Find a node s whose child nodes, n_i , $i = 1, 2, \dots, m$, are all leaf nodes.
If there is no such a node s , then stop.
- Step 2** If s has only one child node n_i , then
Group s , n_i to a new node n'_i so that $e(n'_i) = e(s) + e(n_i)$.
Delete s and n_i from the tree.
Attach y to the place of s .
Go to step 1.
- Step 3** For s and n_i , $1 \leq i \leq m$, do

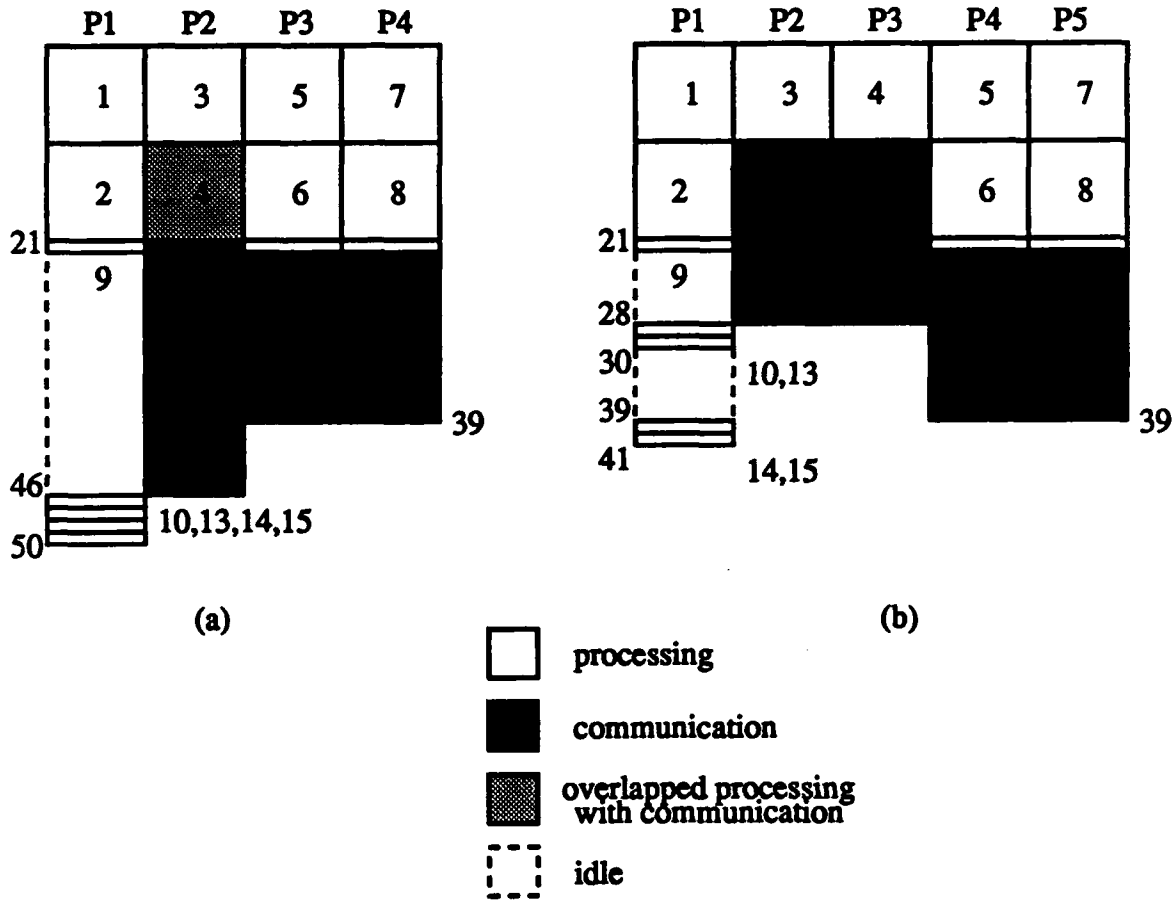


Figure 6.6: A schedule obtained from our approach: (a) for 4 processors, and (b) for 5 processors.

Step 3.1 Calculate the total execution time in one processor.

Let τ be the total execution time. Then,

$$\tau = e(s) + \sum_{i=1}^m e(n_i).$$

Step 3.2 Let $\sigma_i = e(n_i) + c(n_i, s)$, $i = 1, 2, \dots, m$.

Find a node n_j so that σ_j is the largest among σ_i , $i = 1, 2, \dots, m$.

Step 3.3 Find a node n_k so that σ_k is the largest among σ_i , $i = 1, 2, \dots, j-1, j+1, \dots, m$.

Step 3.4 If $\sigma_k > \tau - e(s)$, then

Group s and n_i , $i = 1, 2, \dots, m$, into a new node y so that $e(y) = \tau$.

Delete s and n_i , $i = 1, 2, \dots, m$, from the tree.

Attach y to the place of s .

else

Create a group for each n_i ,

$i = 1, 2, \dots, j-1, j+1, \dots, m$.

Group s and n_j to a new node r

so that $e(r) = \max(e(n_j) + e(s), \sigma_k + e(s))$.
Delete s and $n_i, i = 1, 2, \dots, m$, from the tree.
Attach r to the place of s .

Step 3.5 Go to step 1.

The following theorem indicates that Algorithm 6.2.3 produces an optimal allocation under certain conditions.

Theorem 6.2.1 *Algorithm 6.2.3 produces an optimal allocation when the following conditions are satisfied: 1) No more grouping is required in the sense that the grains of proper size have been obtained. 2) The task precedence graph is a tree. 3) There are more processors than leaf nodes in the task precedence graph.*

Proof. If Condition 1) is satisfied, σ_k as defined in Step 3.2 is always less than or equal to $\tau - e(s)$ in Step 3.4 of Algorithm 6.2.3. Thus, the *else* part of Step 3.4 will always be executed. Conditions 2) and 3) imply that the completion time of the node s (the time required to complete the node s and its child nodes) only depends on how the node s and its child nodes are allocated. Thus, we only need to show that Algorithm 6.2.3 generates an optimal solution for a subgraph of the given task precedence graph.

To do this, let the allocation produced by Algorithm 6.2.3 for s and $n_i, i = 1, 2, \dots, m$, be A , and its completion time be C_A . Then, we assume that there exists another algorithm which generates a better allocation for s and $n_i, i = 1, 2, \dots, m$, be B and its completion time be C_B . In the following, we will show that there exists no allocation B such that $C_B < C_A$: In A , each $n_i, i = 1, 2, \dots, j-1, j+1, \dots, m$, will be allocated to different processors, and in each cycle of Algorithm 6.2.3 only one node is chosen to group with its parent, and the addition of any other nodes to this group will not reduce the completion time by Condition 1). Suppose in B that a node $n_{j'}$, $j' \neq j$, is selected in Step 3.2 and is grouped with its parent s . In Step 3.3, n_j will be chosen instead of n_k because σ_j is the largest among $\sigma_i, i = 1, 2, \dots, m$.

Then, $C_B = \max[\sigma_j + e(s), e(n_{j'}) + e(s)]$. Since we know that $\sigma_j > e(n_{j'})$,

$$C_B = \sigma_j + e(s). \quad (1)$$

From A ,

$$C_A = \max[e(n_j) + e(s), \sigma_k + e(s)] \quad (2)$$

Therefore, two cases need to be considered:

In Case 1), if $e(n_j) + e(s) \geq \sigma_k + e(s)$, then from (2)

$$C_A = e(n_j) + e(s) \quad (3)$$

Comparing C_B in (1) and C_A in (3), we have $C_B - C_A = \sigma_j - e(n_j) = c(n_j, s) \geq 0$. Hence, $C_B \geq C_A$.

In Case 2), if $\sigma_k + e(s) > e(n_j) + e(s)$, then from (2)

$$C_A = \sigma_k + e(s). \quad (4)$$

Comparing C_B in (1) and C_A in (4), we have $C_B - C_A = \sigma_j + e(s) - \sigma_k - e(s) = \sigma_j - \sigma_k \geq 0$ since $\sigma_j > \sigma_k$. Hence, $C_B \geq C_A$. This completes the proof of the theorem.

We show the complexity of Algorithm 6.2.3 in the following theorem.

Theorem 6.2.2 *The time complexity of Algorithm 6.2.3 is $O(n)$, where n is the number of nodes of the task precedence tree.*

Proof. Steps 1, 2 and 3.1 each requires $O(1)$ time to complete each. Steps 3.2 and 3.3 require visiting m nodes, where m is the number of child nodes. Thus, $O(m)$ time is required to complete these steps. Step 3.4 also requires visiting m nodes, and thus $O(m)$ time is needed to complete the step. Once they are visited, each node will not be visited again since they are grouped at this step. Thus, the entire algorithm requires visiting at most once each node. Therefore, the time complexity of this algorithm is $O(n)$. This completes the proof of the theorem.

6.3 Graph Parallelism

Now we extend our grain size determination approach to the general cases in which the task precedence relation is represented by a directed graph. We first define the *depth* and *height* in the directed graph.

Definition 6.3.1 The *depth* of a node in a graph is the length of the longest path from the highest ancestor of that node.

A node having no incoming edge is of depth 0.

Definition 6.3.2 The *height* of a graph is the largest depth of the graph.

A *gain graph* G_g and a *task precedence graph* G_p are defined in the same manner as the *gain tree* and the *task precedence tree* except that they are graphs. In the graph cases, we also analyze gains to determine the proper grain sizes. We first build a gain graph and apply Algorithm 6.2.2 to determine the grain sizes. The following is an algorithm to build a gain graph:

The input to this procedure is a task precedence graph G_p and its output is a gain graph.

Algorithm 6.3.1 Build Gain Graph

- Step 1** Determine the depth of each node in G_p .
Step 2 Initialize all the nodes 'unmarked'.
Step 3 Let the height of G_p be h .
 For a depth $d = 0$ to h , do
 For each node s of depth d , do
 Step 3.1 If $d > 0$, then
 Find the predecessor nodes $n_i, i = 1, 2, \dots, m$, of s
 having a depth $d - 1$.
 If each $n_i, i = 1, 2, \dots, m$, is not 'marked', then
 Call Gain-Analysis(s, n_1, n_2, \dots, n_m).
 Set $n_i, i = 1, 2, \dots, m$, 'marked'.
 If the out-degree of s is greater than 1, then
 set s 'marked'.
 Connect the gain node with the existing
 gain nodes.
 Step 3.2 Find the successor nodes $n'_i, i = 1, 2, \dots, m'$, of s
 having a depth $d + 1$.
 Call Gain-Analysis($s, n'_1, n'_2, \dots, n'_{m'}$).
 Set $n'_i, i = 1, 2, \dots, m'$, 'marked'.
 Connect the gain node with the existing gain nodes.

In Step 1, the depth of each node can be determined using a breadth-first search method. In Step 2, all the nodes are initially set as 'unmarked', meaning that the node is not involved in any grouping. In Step 3, the gain graph is built by visiting each node. First, a node having the smallest depth is chosen, and its predecessor nodes are first checked to determine whether all of them are already included in any group. If they were not included in any group, then *GAIN* is calculated. The predecessor nodes are all set 'marked', and the node is set 'marked' unless it has only one successor node. A gain node representing the nodes under consideration is created. Edges are established from the existing gain nodes to the newly-created gain node if there is a node common to both the new gain node and the existing nodes. The similar steps are applied to the successor nodes.

Theorem 6.3.1 *The time complexity of Algorithm 6.4.1 is $O(n \log n)$.*

Proof. Steps 1 and 2 require visiting each node once, and $O(n)$ time is required to complete the steps, where n is the number of the nodes in the task precedence graph. The complexity of Step 3 is dominated by the procedure Gain-Analysis, and

thus each pass of Step 3 requires $O(m + m')$ time, where m and m' are the number of the predecessor nodes and the number of the successor nodes, respectively. Step 3 needs to be executed for each node in the graph. Because $m + m'$ is the total number of edges in a node, each edge needs to be visited at most twice and thus the time complexity of this step is $O(e)$, where e is the number of the edges in the task precedence graph. Therefore, the time complexity of Algorithm 6.3.1 is $O(e)$. This completes the proof of the theorem.

Once the gain graph is built, we can apply Algorithm 6.2.2 to determine the proper grain sizes. Then, the set of grains obtained by applying Algorithm 6.2.2 can be used as the input for the scheduling stage.

Compared to McCreary's approach [54], our approach has the following advantages:

First, our approach is more efficient in terms of the time complexity. As shown above, the time complexity of our approach is $O[\max(n \log n, e)]$ in which $n \log n$ is for Algorithm 6.2.2 and e is for Algorithm 6.3.1. Here, n and e represent the number of nodes and the number of edges in the task precedence graph, respectively. The time complexity of McCreary's approach is dominated by the parsing algorithm of $O(n^3)$.

Second, our approach is more general in the sense that any acyclic task graph can be analyzed for grain size determination. McCreary's method can handle only regular dependency relations among the nodes, such as the case where the left and right child nodes have symmetric dependency relations to their parents.

Third, our approach considers the execution time of the nodes and the communication time between the nodes as primary factors from the beginning of the grain size analysis and selects the best candidates from the entire task precedence graph based on the analysis of such information. McCreary's method, however, may lose the opportunity of grouping at later stages because the decomposition of the graph is done without using such information.

To illustrate our approach we use the example used in [54]. The task precedence graph for the FFT (Fast Fourier Transformation) is shown in Figure 6.7. Using Algorithm 6.3.1, we obtain the gain graph shown in Figure 6.8. We also show a schedule for the FFT problem in the case of four processors in Figure 6.9.

6.4 Pipelined Parallelism

In exploiting pipelined parallelism, one important consideration is how to divide a program into a set of tasks to reduce the completion time of the program. We call each task in the pipelined program as a *segment*. In the following, we show that we can find the optimal size of segments. Note that the optimal segment size correspond to the optimal grain size.

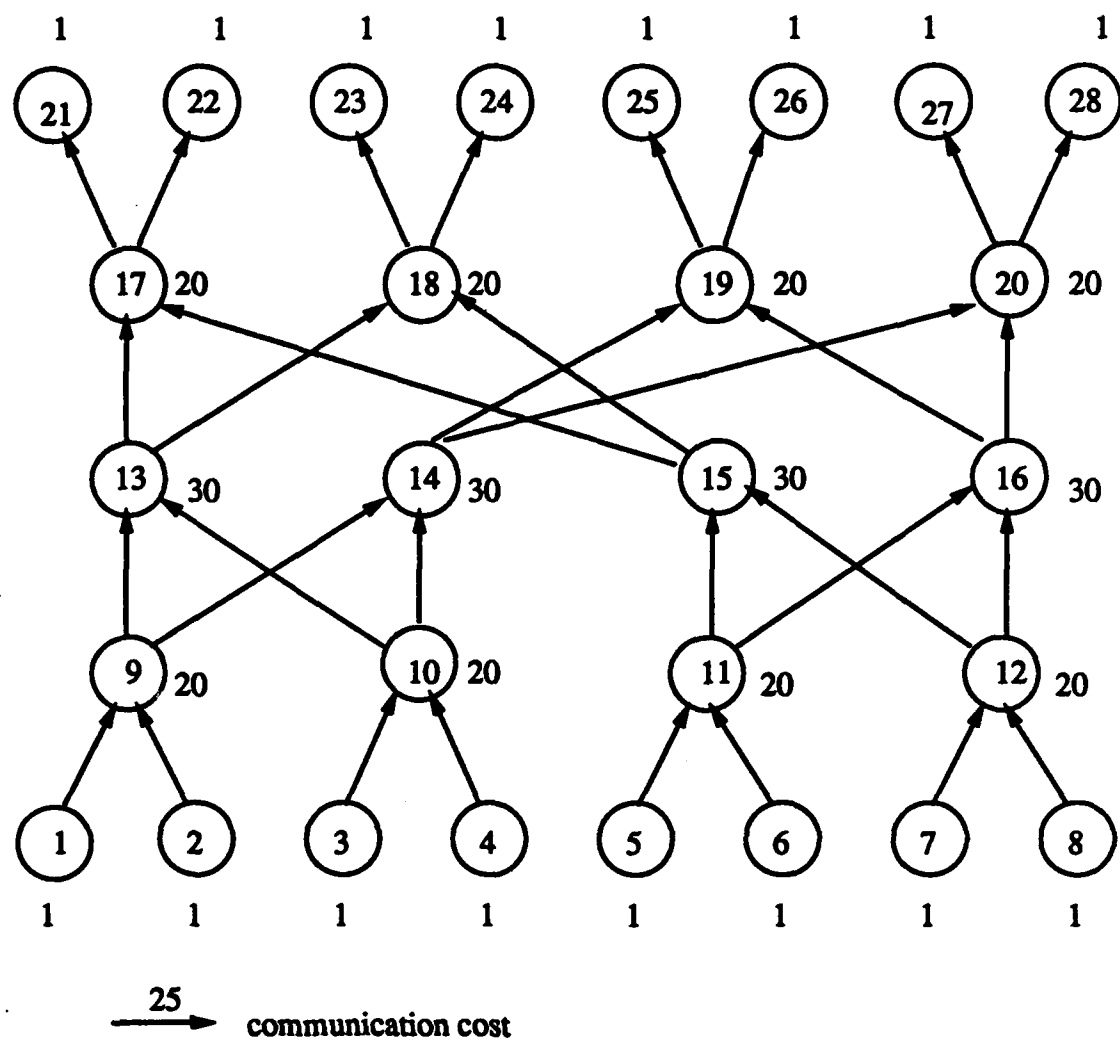


Figure 6.7: A task precedence graph for the Fast Fourier Transformation problem.

Definition 6.4.1 The *dominant segment* in a pipelined program is a task such that its completion time is the largest among all the tasks in the program.

The dominant segment dictates the completion time of the entire pipelined program because the processor for executing the dominant segment in a pipelined program is always busy once all the segments are filled with data.

Definition 6.4.2 All the tasks other than the dominant segment are called *subordinate segments*.

Definition 6.4.3 The *computation segment* in a pipelined program is a task which receives input data, executes the computation associated with the segment and returns the result.

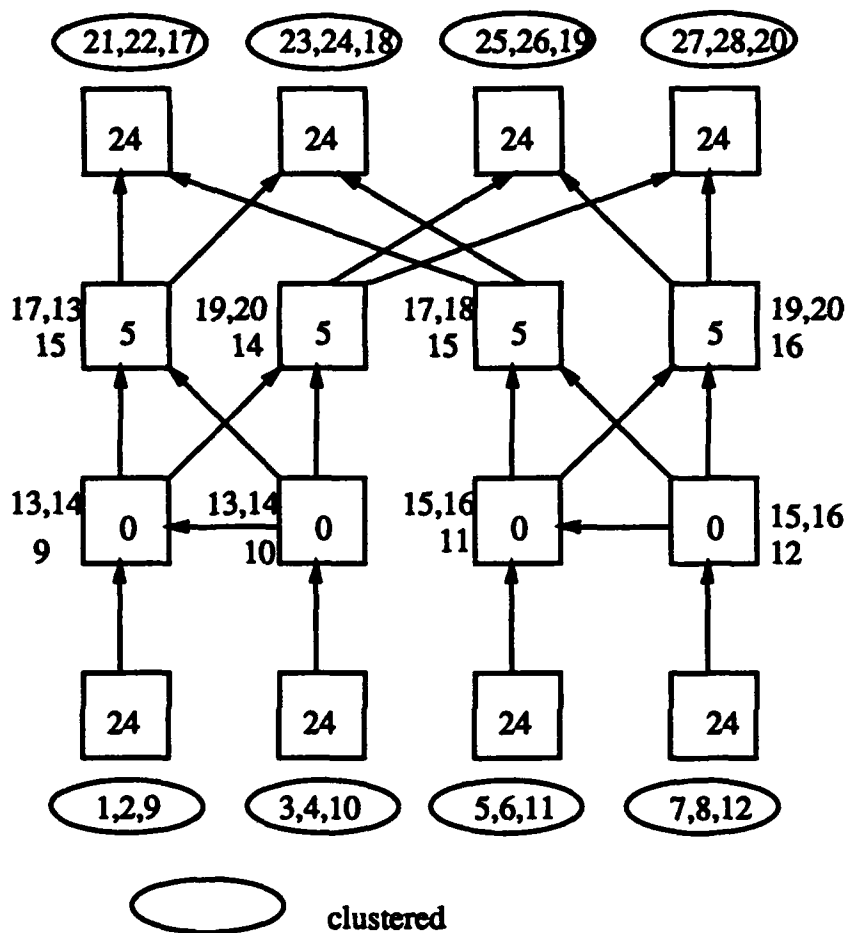


Figure 6.8: A gain graph for the Fast Fourier Transformation problem.

Definition 6.4.4 The *communication segment* in a pipelined program is a task which passes data from the preceding computation segment and to the succeeding computation segment.

Definition 6.4.5 The *one-pass completion time* $E(\Phi_i)$, defined for a segment Φ_i of a pipelined program, is the amount of time required to complete processing of a datum in Φ_i .

Note that the dominant segment can be either a computation segment or a communication segment, but the neighbors of a communication segment are always computation segments and vice versa. Computation segments and communication segments have different properties. Computation segments can be segmented according to the data dependency relations among the segments to reduce the one-pass completion time. However, communication segments cannot be so refined. This distinction implies that in order to find the optimal grain sizes, we need to begin the grain size analysis from the smallest grain size available. In our approach, the IPR presents the smallest grain parallelism we can get in the program statement level.

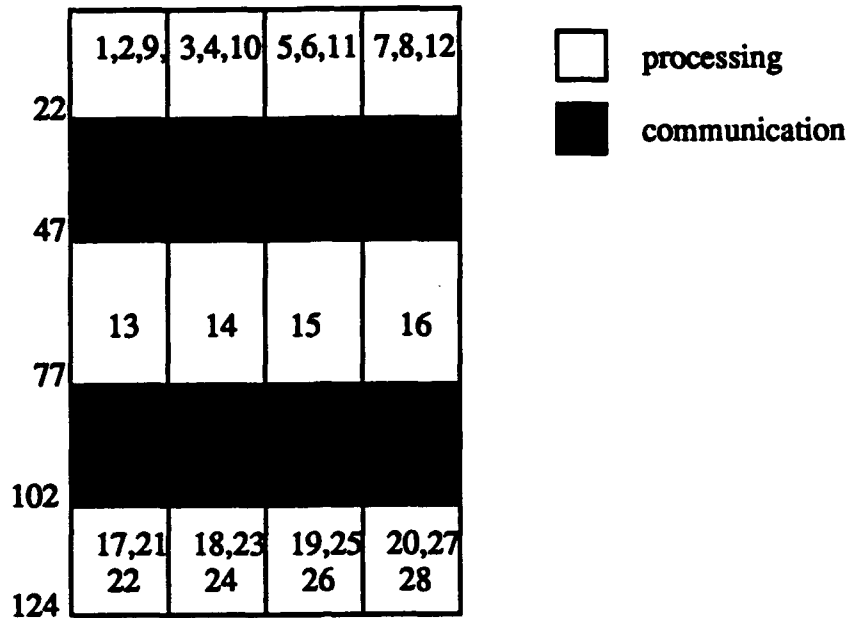


Figure 6.9: A schedule for the FFT problem.

The goal of finding proper grain sizes in a pipelined program is to reduce the one-pass completion time of the dominant segment by either combining a dominant segment with its neighboring segments or refining a dominant segment. However, since the IPR presents as fine grain parallelism as possible, the refinement of the dominant segment would not be considered. In the following, we will present an optimal grain size determination algorithm for the pipelined parallelism in the sense that the completion time of a pipelined program is minimized for all allocations.

The input to this algorithm is the execution time of the computation segments in a pipelined program and the communication time of the communication segments, and its output is an optimal grain size. We make the following assumptions on parallel processing systems:

- The one-pass completion time for each segment is fixed and known a priori.
- The amount of time for initialization of the segments in a pipelined program is negligible.
- The execution of code and communication can be done simultaneously.

It is noted that these assumptions do not impose substantial restrictions on the applicability of our approach for grain size determination in pipelined programs. The first assumption is realistic in that we can easily obtain the one-pass completion time by

performing an analysis on the assembly language code of the pipelined program. In a large pipelined program, the amount of time for the initialization of the segments in the pipelined program can be ignored because it is too small when it is compared to the amount of time for which all the segments in the pipelined program are active. Thus, the second assumption is valid. Due to the existence of DMA (Direct Memory Access) chips in parallel processing systems, such as transputers, the execution of the code and communication can be done simultaneously. Hence, the third assumption is also valid.

Algorithm 6.4.1 *Determine Grains for Pipelined Parallelism*

- Step 1** Sort the segments using one-pass completion time as a key in the descending order.
 Let $\Phi_i, i = 1, 2, \dots, n$, be the segments in the program.
- Step 2** Find a dominant segment $\Phi_j, 1 \leq j \leq n$.
 Let the preceeding and the succeeding segments of Φ_j be Φ_{j-1} and Φ_{j+1} , respectively.
- Step 3** If Φ_j is a communication segment, then
 If $E(\Phi_{j-1}) + E(\Phi_{j+1}) < E(\Phi_j)$, then
 Group Φ_{j-1}, Φ_j and Φ_{j+1} to a new computation segment $\Phi_{j'}$
 so that $E(\Phi_{j'}) = E(\Phi_{j-1}) + E(\Phi_{j+1})$.
 Delete Φ_{j-1}, Φ_j and Φ_{j+1} from the list.
 Go to Step 2.
 else
 Stop.

We will show in the following theorem that Algorithm 6.4.1 generates the optimal grain sizes for pipelined parallelism.

Theorem 6.4.1 *Algorithm 6.4.1 generates the optimal grain sizes.*

Proof. When the dominant segment is a computation segment since we cannot divide the computation any further, the grains cannot be further grouped. Thus, the dominant segment remains the same, and so does the completion time of the entire program. Now consider a case in which the dominant segment is a communication segment. In this case, the algorithm always tries to group the dominant segment with its adjacent computation segments to reduce the one-pass completion time.

To prove this theorem, let a solution for a pipelined program obtained by Algorithm 6.4.1 be S_1 . Assume that there exists a solution for the pipelined program, called S_2 , obtained by another algorithm, and the completion time by S_2 is smaller than the completion time by S_1 . In the following we will show that S_2 cannot exist. Suppose that Φ_j is the dominant segment such that

$$E(\Phi_j) > E(\Phi_k), \text{ for } 1 \leq j \leq n, 1 \leq k \leq n \text{ and } j \neq k. \quad (5)$$

Suppose that in S2 a communication segment Φ_k is grouped with its adjacent segments Φ_{k+1} and Φ_{k-1} before the dominant segment Φ_j is grouped with its adjacent segments Φ_{j+1} and Φ_{j-1} . When Φ_j and Φ_k are not adjacent communication segments, grouping of Φ_k with its adjacent segments will not affect grouping Φ_j with its adjacent segments. Thus, we need to consider only the case in which Φ_j and Φ_k are the two adjacent communication segments.

Grouping of Φ_k with its neighbors Φ_{k+1} and Φ_{k-1} implies that

$$E(\Phi_{k-1}) + E(\Phi_{k+1}) < E(\Phi_k). \quad (6)$$

From (5) and (6), we have

$$E(\Phi_{k-1}) + E(\Phi_{k+1}) < E(\Phi_j). \quad (7)$$

When Φ_j and Φ_k are two adjacent communication segments in S2, it may not be possible to group Φ_j with Φ_{j-1} and Φ_{j+1} ($= \Phi_{k-1}$). If grouping of Φ_j with Φ_{j-1} and Φ_{j+1} is not possible, S2 may not be an optimal solution because such grouping may be feasible in S1.

Now, we would like to show that there is a case where in S2 Φ_j cannot be grouped with Φ_{j-1} and Φ_{j+1} , but in S1 Φ_j can. Consider the case of $E(\Phi_{j-1}) + E(\Phi_{k-1}) < E(\Phi_j) < E(\Phi_{j-1}) + E(\Phi_{k-1}) + E(\Phi_{k+1})$. Since this case does not violate (7), it is considered as a legal case. Then, from $E(\Phi_j) < E(\Phi_{j-1}) + E(\Phi_{k-1}) + E(\Phi_{k+1})$, in S2 Φ_j will not be grouped with its two adjacent segments because such grouping will increase the one-pass completion time of the segment. Consequently, the completion time of the pipelined program in S2 is determined by $E(\Phi_j)$. On the other hand, from $E(\Phi_{j-1}) + E(\Phi_{k-1}) < E(\Phi_j)$, in S1 Φ_j will be grouped with its two adjacent segments by Step 3 of Algorithm 6.4.1, and then the completion time of the pipelined program is determined by $E(\Phi_{j-1}) + E(\Phi_{k-1})$, which is smaller than $E(\Phi_j)$. Thus, we show that there is a case such that S2 cannot reduce the completion time of the pipelined program, but S1 can. Therefore, S2 cannot exist. Hence, we have shown that Algorithm 6.4.1 can always generate an optimal grain sizes. This completes the proof of the theorem.

Theorem 6.4.2 *The time complexity of Algorithm 6.4.1 is $O(n \log n)$, where n is the number of segments of a pipelined program.*

Proof. Step 1 requires $n \log n$ steps. Using a priority queue data structure to store the sorted list, we can retrieve and store any element in $\log n$ steps. Thus, each execution of Step 3 requires at most $4 \log n$ steps for three deletions and one addition to the priority queue. Since there are at most n segments, the entire algorithm can run in $O(n \log n)$ time. This completes the proof of the theorem.

6.5 Modifying Intermediate Form

The IPR obtained from the front-end transformation consists of only a set of nodes which have not been subject to grain size analysis. Before transforming the IPR to the target code, the IPR is modified to reflect the information obtained from the partitioning and the grain size analysis. This process is called *Modifying Intermediate Form* and consists of the following two steps:

- Modification of the IPR using partitioning information.
- Modification by performing grain size analysis.

The modified IPR is called *Modified Intermediate Form* (MIF) which is a hierarchical intermediate form that is architecture dependent. It consists of the following 3 levels of representation:

- **Primitive node level (P-Graph):** This represents a unit of code corresponding to the body of a control thread in a processor. The P-Graph consists of a set of primitive nodes called *P-Nodes* and a set of edges connecting them. P-Nodes are the smallest computational unit in the intermediate form, such as +, -, *. The edges represent data dependency.
- **Macro Graph level (M-Graph):** This represents the multiple threads or process level parallelism in a processor. The M-Graph consists of a set of nodes called *M-nodes* or *macro nodes* and a set of edges connecting them. An edge between two nodes represents the existence of communication between the two nodes. Each macro node has a P-graph corresponding to it.
- **File Graph level (F-graph):** This represents the processor level parallelism in the application. The F-Graph consists of a set of nodes called *file nodes* or *F-nodes* and a set of edges connecting them where the edges represent the communication between the two file nodes. Each F-node has a M-Graph corresponding to it.

While being executed on the parallel processing system, the parallel program must be separated into a set of individual programs with acceptable grain size. Allocation must then be performed on it based on the number of processors available for the program. There are two types of parallelism: software parallelism and hardware parallelism. Software parallelism is said to be achieved when multi-threading on a processor is achieved. It is actually the hardware parallelism which gives one substantial gain in performance. Software parallelism is achieved at the M-Graph level. Furthermore, the M-graph is subject to clustering and we obtain a set of File nodes called F-nodes, where each File node consists of a set of M-nodes. This set of M-nodes is called a M-Graph. The M-nodes in a M-Graph can be executed as parallel processes on a processor.

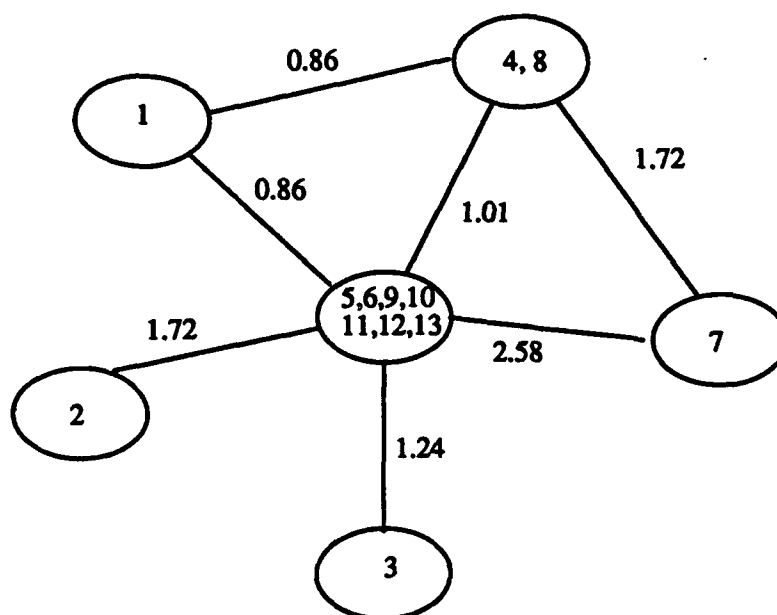


Figure 6.10: An example illustrating the output of partitioning.

The F-graph is a graph consisting of a set of F-nodes which gives the number of files to be generated. Each F-node is executed on a different processor. The F-graph is directly mapped on to the processors. The F-Nodes are numbered 1 to N where N is bounded by the number of processors. It can be seen that with the modified intermediate form we can partition the program into a set of executable modules where each module can be executed on a transputer. The macro nodes can become the different threads executing concurrently on the same processor. The file identification numbers are analogous to the processor identification numbers.

To illustrate the above feature let us consider a simple example whose modified intermediate forms consists of two F-nodes and each F-node corresponds to four M-nodes. In this case we can generate two files to be run on two processors, each having four concurrently executable threads. Let the two files be F_0 and F_1 . The two files F_0 and F_1 can be mapped on to two processors T_0 and T_1 . Assume that there is communication between T_0 and T_1 and that T_0 sends 200 bytes of message to T_1 . This is equivalent to saying that there exists communication between F_0 and F_1 . We can also specify which M-node in F_0 is the source of the communication and which M-node on F_1 is the destination for the message. In such a case we have all the information required for communication between two processes, that is the destination processor and the particular process on the processor to which the message is intended. We can thus get a one-to-one mapping of the MIF into various processes executing on the processors, such as a network of transputers.

6.5.1 Modifying Intermediate Form Using Partitioning Information

We will illustrate our approach to modify the intermediate form with an example. Consider Figure 6.10 which shows an example output of the partitioning stage.

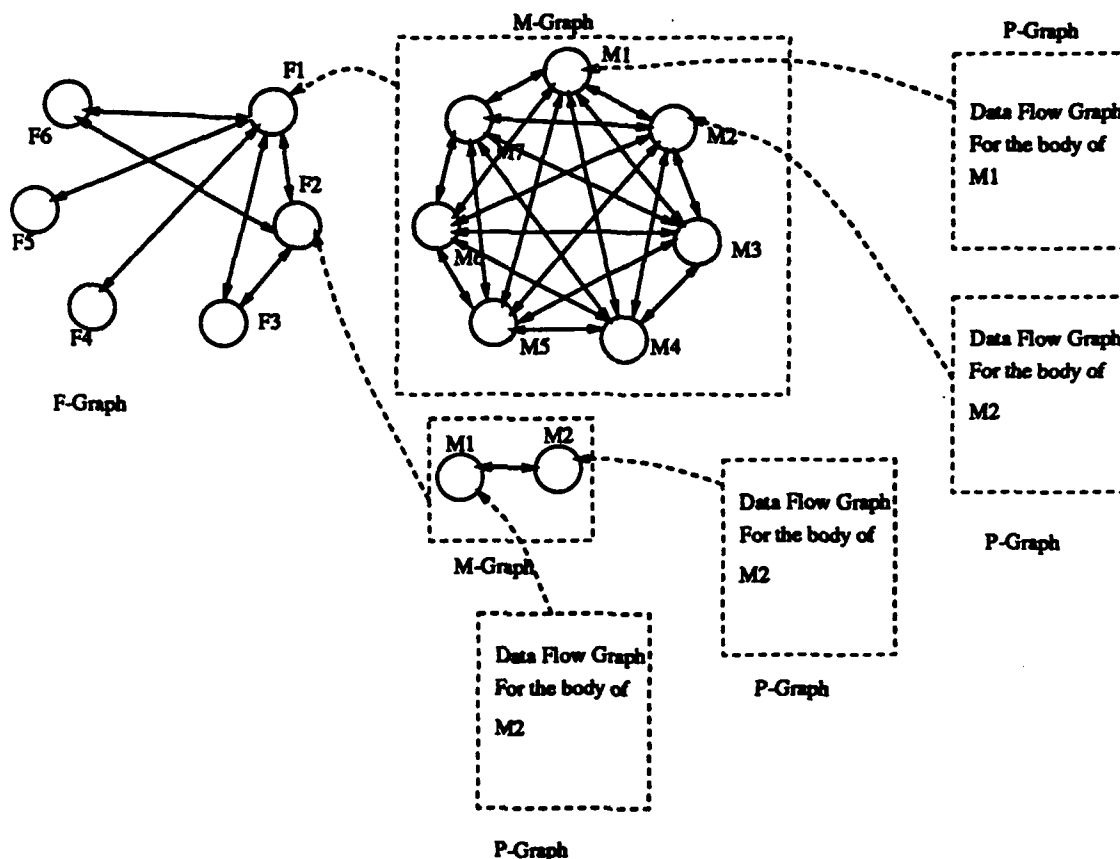


Figure 6.11: A modified intermediate form graph based on partitioning.

Figure 6.11 shows six clusters and two of those clusters have more than one object. Let us assume that each of these clusters is executed on a single processor. This results in more than one control thread in two of the processors and is represented by more than one macro node corresponding to single file node for those processors. The P-Graph which consists of primitive nodes are executed serially. Hence, there is no overhead due to communication while executing the P-Graph.

6.5.2 Modifying Intermediate Form Using Grain Size Analysis

We will now illustrate our method by incorporating the information resulting from grain size analysis in the IPR. Figure 6.3 shows a task precedence graph. We will use our schedule shown in Figure 6.6 (a) for this task precedence graph to modify the IPR. We see that tasks 1,2,9,10,13,14 and 15 are executed on processor P1. These tasks are to be executed sequentially to remove the overhead due to communication and is shown in Figure 6.12. There is one macro node corresponding to each file node and each of the macro nodes has their corresponding P-Graph. The P-Graph corresponding to the M-node of F-node F1 shows the precedence relation among the tasks 1,2,9,10,13,14, and 15. The P-Graph corresponding to the M-node of F-node F2 shows the precedence between the tasks 3 and 4. Similarly, we obtain the graphs corresponding to the other F-nodes shown in Figure 6.12. Grain size determination

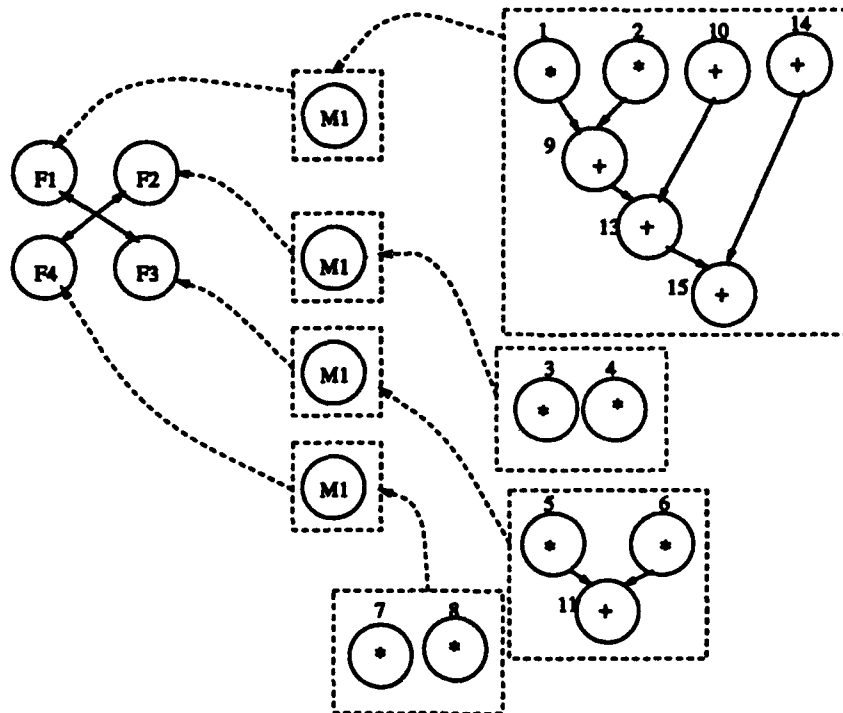


Figure 6.12: A modified intermediate form after grain size determination

is used to exploit the parallelism within a cluster of objects which are the result of partitioning. Since F-nodes represent the parallelism among the tasks, we can allocate a cluster over more than one F-node to represent the parallelism within a cluster.

Chapter 7

PROOF/L Back-end Translation

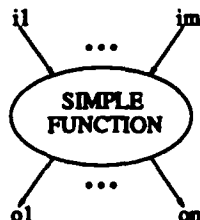
In this chapter, we will present the translation rules for translating the program in IPR to Inmos C of a transputer which is our target language. We also identify schemes for detecting iterations and predicates in the IPR that can be translated into a suitable control statement in C. Finally, we will illustrate our translation rules with examples.

7.1 Translation Rules

7.1.1 Simple Structure

We have identified eleven types of nodes in the IPR: simple function, identity function, constant function, copy function, macro function, latch, selector, distributor, merge, construct, and split. The translation rules for each of these types of nodes are given as follows:

1. *simple function*



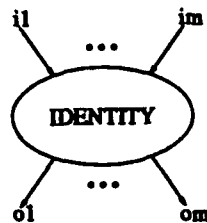
A *simple function node* represents a primitive function such as $*$, $-$, $+$, \dots , including boolean and logic operators and a user defined function. Simple functions can be translated based on the number of input and output parameters required. The number of input and output parameters can be easily determined by analyzing the textual representation of the IPR. The translation rule can be expressed as follows:

For a unary operator,
 $o = \text{unop } i;$

For a binary operator,
 $o = i_1 \text{ binop } i_2;$

For a user-defined function,
 $V(i_1, \dots, i_m, o_1, \dots, o_n);$

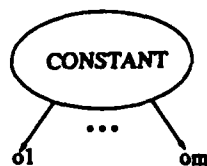
2. identity function



An *identity node* is translated so that the input is directly returned as the output. The translation rule can be expressed as follows:

$$o_1 = i_1; o_2 = i_2; \dots o_m = i_m;$$

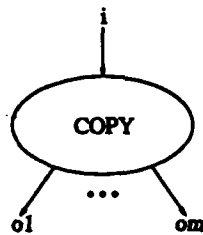
3. constant



A *constant node* is translated so that the specific same value is produced all the time. If the value of node is numeric, then this node is a constant node. The translation rule can be expressed as follows:

$$o_1 = \text{Const}; o_2 = \text{Const}; \dots o_m = \text{Const};$$

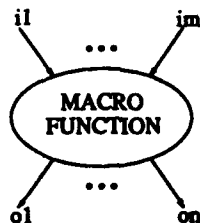
4. copy



A *copy node* is translated so that the appropriate number of copies having the same value as that input are produced. That is, it transfers its input value onto all of its output edges. The translation rule can be expressed as follows:

$$o_1 = i; o_2 = i; \dots o_m = i;$$

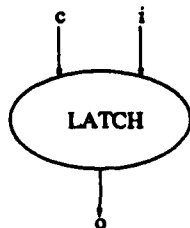
5. macro function



A *macro function* represents a compound function composed of simple and/or macro functions. This function creates a process for parallel execution. The translation rule can be expressed as follows:

$$p_i = V(\text{Process } *p, \# \text{ of parameters, } i_1, \dots, i_m, o_1, \dots, o_n);$$

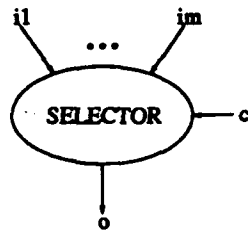
6. latch



A *latch node* is for sequential operations. The input data is transferred onto the output edge when the control is fired. The translation rule can be expressed as follows:

$$o = i;$$

7. selector



A *selector node* represents a conditional construction function by combining other nodes. Conditional constructions achieve selective routing of data among inputs. Boolean or index-valued data can be produced by a node that performs some decision function. This node with related nodes is translated so that one of the inputs as an output according to the value of control data is returned. The translation rule can be expressed as follows:

- if $m = 1$,


```

      if (c)
      {
        o = i1;
      }
      
```
- if $m = 2$,

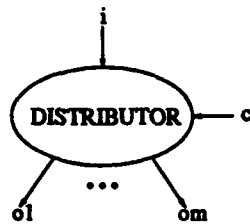

```

      if (c)
      {
        o = i1;
      }
      else{
        o = i2;
      }
      
```
- Otherwise,


```

      switch (c)
      {
        case 1: o = i1;
        case 2: o = i2;
        ...
        case m: o = im;
      }
      
```

8. distributor



A *distributor node* is used in conditional constructs. It is translated so that the input is passed to one of the output ports according to the value of control data. The translation rule can be expressed as follows:

- if $m = 1$,


```

      if (c)
      {
          o1 = i;
      }
      
```
- if $m = 2$,

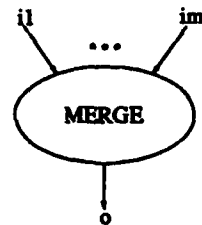

```

      if (c)
      {
          o1 = i;
      }
      else{
          o2 = i;
      }
      
```
- Otherwise,


```

      switch (c)
      {
          case 1: o1 = i;
          case 2: o2 = i;
          ...
          case m: om = i;
      }
      
```

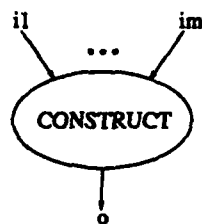
9. merge



A *merge node* represents a nondeterministic selector, which receives an arbitrary number of input data at a time and returns one, which arrives first to it. If a merge node has more than one available inputs at the same time, one of the inputs is chosen as the output arbitrarily or by priority. The translation rule can be expressed as follows:

$$o = \text{One of } (i_1, \dots, i_m);$$

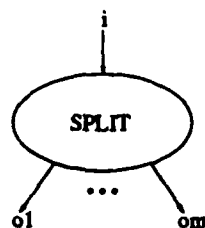
10. construct



A *construct node* receives one or more input values and make it as a list. The translation rules are given as follows:

$$o_1 = i_1; o_2 = i_2; \dots o_m = i_m; \quad o \text{ is } \text{list}(o_1, o_2, \dots, o_m).$$

11. split



A *split node* receives a list as an input and split that list into values. The translation rule can be expressed as follows:

$$o_1 = i_1; o_2 = i_2; \dots o_m = i_m; \quad i \text{ is } \text{list}(i_1, i_2, \dots, i_m).$$

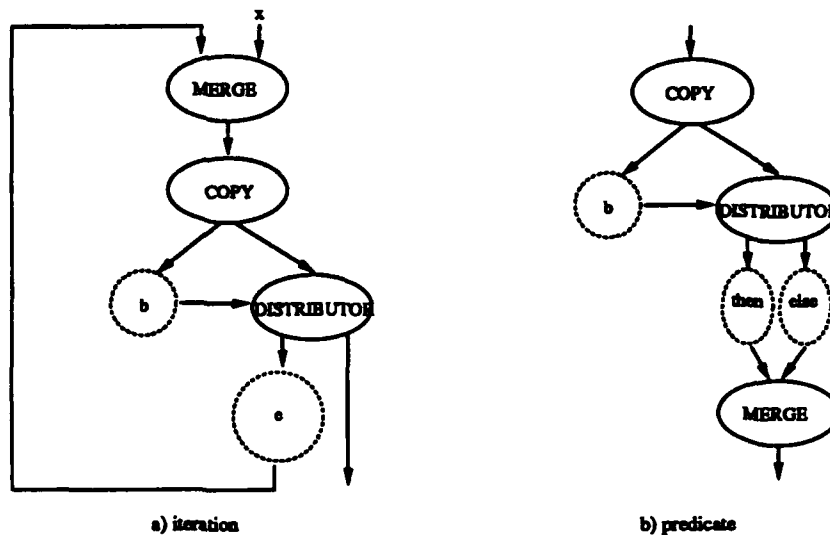
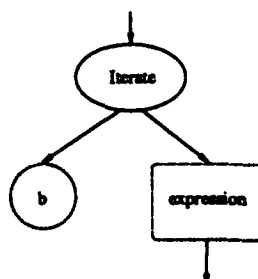


Figure 7.1: The control structures for *iteration* and *predicate* statements.

7.1.2 Schemes for Detecting Patterns

High level languages support statements such as *iteration* and *predicate* statements and it is therefore advantageous to detect patterns in the IPR that can be translated into such high level language control structure statements and to reconstruct the IPR. Iteration can be achieved through cyclic data flow graph in IPR. The body of the iteration is initially achieved by a datum that arrives on the input of the graph. The body produces a new datum, which is cycled back on a feedback path until a certain condition is satisfied. A predicate in IPR can be achieved by receiving the value of boolean expression which is either *true* or *false*. It determines from which of the two outputs the data is available, either *then_part* or *else_part*. They are shown in Figure 7.1. The process of detecting these patterns starts when the translator encounters a “copy” node and tries to detect the related nodes. Then, iteration and predicate structures are reconstructed and translated as follows:

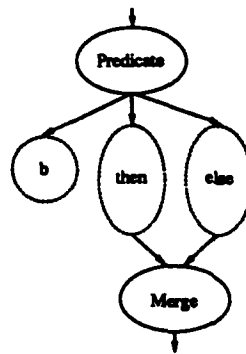
- *Iteration.*



The above iteration control structure can be translated as follow:

```
while (b)
{
    expression;
}
```

- *Predicate*



The above predicate control structure can be translated as follow:

```
if (b)
{
    then - expression;
}
else{
    else - expression;
}
```

7.2 An Algorithm Traversing the Data Dependency Graph

The translator starts translating IPR to a target language when a node with no predecessor is first reached. That is, the algorithm to traverse the graph proceeds by translating a node in the graph that has no predecessor. Then, this node together with all edges leading out from it is deleted from the graph. These two steps are repeated until all the nodes are translated. An algorithm for traversing and translating each node is shown as follows:

```

    // Input: Intermediate Program Representation  $G = (V, E)$ 
    // Output: Target code executable on a transputer
1.  $P \leftarrow \emptyset$ ;
2. while  $\|V\| > 0$  do
    begin
3.     if no node  $v \in V$  with no predecessor then
4.         begin // parallel execution
5.             if  $P$  is empty then
6.                 begin
7.                     if  $\|V\| = 0$  then return
8.                     else error
9.                 end
10.            end
11.         else
12.            begin
13.                translate all  $v \in P$  for parallel execution;
14.                remove  $(v, w)$  from  $E$ 
15.                remove  $v$  from  $V$ 
16.                 $P \leftarrow \emptyset$ ;
17.            end
18.        end
19.    end
20. else
21.    begin
22.        if  $v \in V$  is a macro function then add  $v$  to  $P$ 
23.    else
24.        begin
25.            translate  $v$ ;
26.            remove  $(v, w)$  from  $E$ 
27.            remove  $v$  from  $V$ 
28.        end
29.    end
30. end
end

```

The above algorithm uses three sets: V , E and P . V contains the vertices of the given graph G . E contains the edges of the graph G . The algorithm works by translating a vertex in G into the target code. P is used to collect the vertices for parallel execution.

We can view the algorithm as a sequence of operations that manipulate the three sets V , E and P . Line 1 initializes the set P . Line 2, which controls the main loop of this algorithm, requires maintaining a count of the number of vertices in the set V . In line 3 we determine whether the translation for parallel execution is selected. Lines 7-10 represent the translation of all v 's in P for parallel execution and P is cleared. In line 11 we determine whether the node is selected for parallel execution and put it into set P . Lines 12-14 represent the translation of simple function or non-computational nodes.

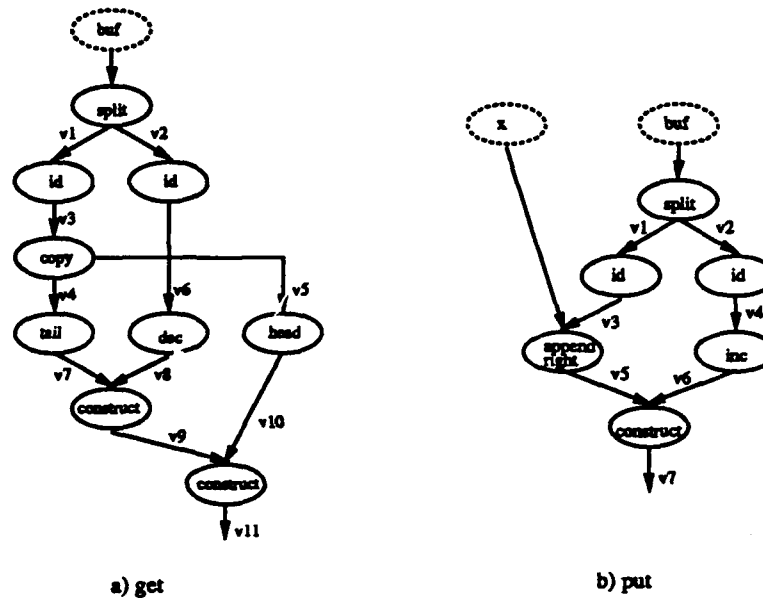


Figure 7.2: The IPR of *get* and *put* methods.

In this implementation two stacks are used. One is for maintaining the list of nodes with zero count, the other stack is used for maintaining the parallel execution. The deletion of all edges leading out of a node can be carried out by decreasing the predecessor count of all nodes adjacent to it. Whenever the count of a node drops to zero, that node can be placed onto a list of nodes with zero count which is maintained as a stack.

7.3 Examples

We will illustrate the back-end translation rules with two examples: one is a Bounded Buffer which is communication-oriented and the other is to compute the factorial which is computation-oriented.

In the Bounded Buffer program, the methods *get* and *put* will be translated as procedures *get* and *put* while implementing in Inmos C code of a transputer.

The graphical representation of the IPR for the methods *get* and *put* is shown in Figure 7.2. The *get* method in IPR is started by introducing an object "buf". The *split* node is now fireable and splits "buf" into two elements, "store" and "count". The data passes through the *id* node and the *copy* node is used to duplicate the data "store". There are no edges connecting the *tail*, *head*, and *dec* nodes and this implies that there is no data dependency between these nodes. That is, these nodes can be executed in parallel. The results are constructed into a list by the *construct* nodes. The *put* method follows similarly.

The following are the high level description of the Inmos C code generated for *get* and *put* methods. The Inmos C code produced here is an unoptimized version.

```

get (process *p, struct buffer *buf, struct buffer *out1, int *out2)
{
    declarations of process and local variables;

    v1 = buf->store;
    v2 = buf->count;
    v3 = v1;
    v6 = v2;
    v4 = v3;
    v5 = v3;
    parallel execution of
        tail(v4, v7);
        dec(v6, v8);
        head(v5, v10);
    v9->store = v7;
    v9->count = v8;
    out1 = v9;
    *out2 = v10;
}

```

```

put (process *p, struct buffer *buf, int x, struct buffer *out1)
{
    declarations of process and local variables;

    v1 = buf->store;
    v2 = buf->count;
    v3 = v1;
    v4 = v2;
    parallel execution of
        append_right(x, v3, v5);
        inc(v4, v6);
    out1->store = v5;
    out1->count = v6;
}

```

For the factorial example, the IPR is shown in Figure 7.3. The factorial program is started by introducing an integer "i". The data "i" of type integer is duplicated and given as inputs to the boolean expression part and the *distributor* node. The output of the *eq* node is a boolean value that indicates whether the data "i" passes then-part

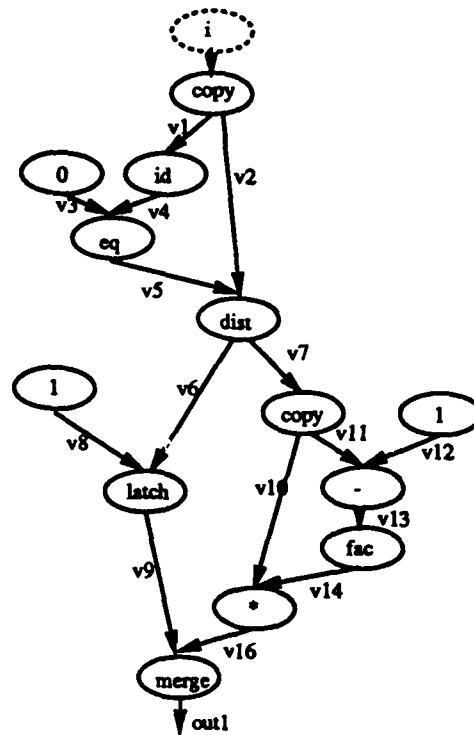


Figure 7.3: The IPR for *factorial*.

or *else_part*. If *true* is produced by *eq*, the value "1" is passed through as the result of the *factorial* method. If the *eq* node produces *false*, the data "i" is passed through the *copy* node and the factorial function is recursively called with the input data as "i - 1". At this point the data "i" and the result of the called factorial function are multiplied and returned as the result of the *factorial* method.

The following is the high level description of the Inmos C code generated for the *factorial* procedure.

```

fac (process *p, int i, int *out1)
{
    declarations of process and local variables;

    v3 = 0;
    v8 = 1;
    v12 = 1;
    v1 = i;
    v2 = i;
    v4 = v1;
    if (v3 == v4) {
        v6 = v2;

```

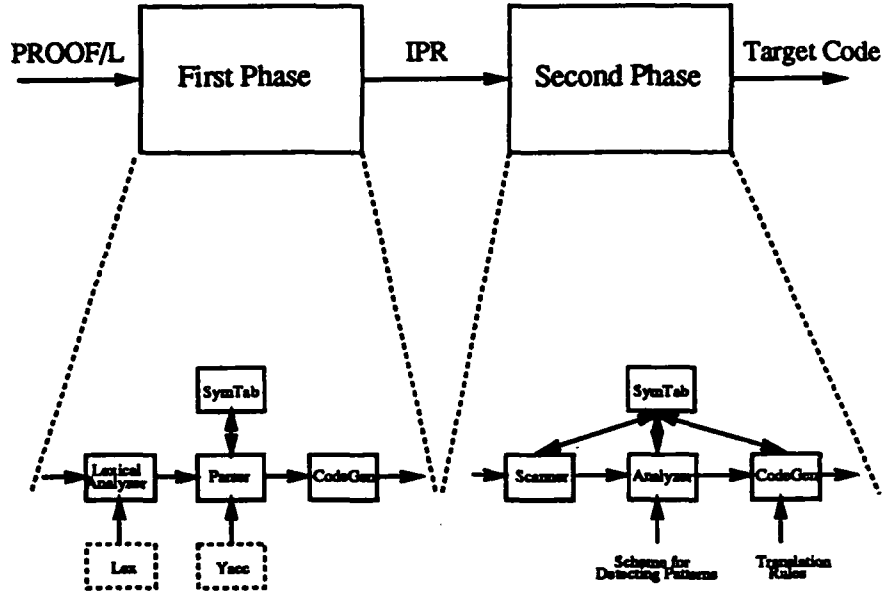


Figure 7.4: The implementation scheme of a translator for PROOF/L to the target code.

```

    v9 = v8;
    *out1 = v9;
}
else {
    v7 = v2;
    v10 = v7;
    v11 = v7;
    v13 = v11 - v12;
    fac(v13, v14);
    v16 = v10 * v14;
    *out1 = v16;
}
}

```

7.4 Implementation of the Back-end Translator

To translate the PROOF/L code to a target language, we developed a two-phase translator. The front-end translator transforms the PROOF/L code to IPR and the back-end translator transforms the IPR to the target language Inmos C. Figure 7.4 depicts the two-phase translation scheme of a PROOF/L program.

The back-end translator receives the IPR as the input and generates the Inmos C code that can be executed on a transputer system as the output. This phase of the translator mainly consists of the following modules: reading the input from the front-end translator, and detecting the data dependency in the data dependency graph, visiting each node in data dependency graph without predecessor nodes, translating such a node into the target code using a suitable translation rule. In the following section, the word "translator" implies the back-end translator.

7.4.1 Reading the Input

Before translating the IPR to a target language, the translator preloads keywords in Inmos C and reserved variable names which are used for local variables, output parameter variables, channel variables, and process variables. The lexeme and token representations for all the keywords are stored in the array `keywords`, which has the entries consisting of a pointer to the `lexemes` array and an integer denoting the type of keywords stored. The operation `init_symbol` inserts the keywords into a symbol table and returns the symbol table index for the lexeme. Operations on reserved variables are similar to that of keywords, but the operations are `init_local`, `init_out`, `init_channel`, and `init_process`, respectively.

The next thing that should be done in the translator is to read the input data generated by the first phase of the translator. The input is represented in textual form and as a UNIX file which consists of a sequence of intermixed data types – characters, integers, and special symbols. It consists of five parts: class declaration, methods, passive object, pseudo active object and active object.

In this implementation, the translator reads the IPR using the C++ stream input and output operations. The `iostream` library in C++ predefines a set of operations for handling reading and writing of the built-in data types. Furthermore, file manipulation using the input and output operations is also supported. To link streams to files, both the header files `iostream.h` and `fstream.h` must be included. To open a stream attached to a file for input, the translator uses the function `open()`. For example,

```
#include <iostream.h>
#include <fstream.h>
ifstream ifile;
char lexbuf[MAXSTRING];
main(int argc, char *argv[])
...
    ifile.open(argv[argc-1], ios::in);
```

Then, to read the input from the initialized stream, the translator uses `ifile >>`
...

```
ifile >> lexbuf;
```

When the translator is reading the input, it distinguishes the input stream depending on keywords – Class, Method, Passive, PActive(Pseudo Active), Active. It collects the input streams and tries to build a data dependency graph of each object and method. Every node in IPR should be represented by its name which corresponds to the operation performed by the node, the number of its predecessors, its set of predecessors, and its set of successors. The two sets are conveniently organized as linked lists. Consequently, an additional entry in the description of each node contains the link to the next node in the list. Analogously, the set of each node's predecessors and successors is conveniently represented as linked lists. Each element of the predecessor list and the successor list is described by an identification, local variable, and a link to the next element on this list. These data structures are shown in Figure 7.5. If we call the data structures of the node list *Node* and the data structures of elements on the predecessor and successor chain *InOut*, we obtain the following declarations of data types:

```
class Node {
    char      *name;
    int       NofInput;
    class InOut*input;
    class InOut*output;
    class Node *next;
};

class InOut {
    char      *id;
    char      *LocalVar;
    char      *LocalType;
    class InOut*next;
};
```

The translator transforms the input data into a linked list structure to build a data dependency graph. This is performed by successively reading the input and generating the node structure, its input and output structures for each node. These structures must be inserted in the list. Subsequently, a new entry is added in the list. At this time the translator assigns a local variable to each output structure so that it can be used in code generation stage.

When it encounters EOF, it finishes reading the input and goes to the next step.

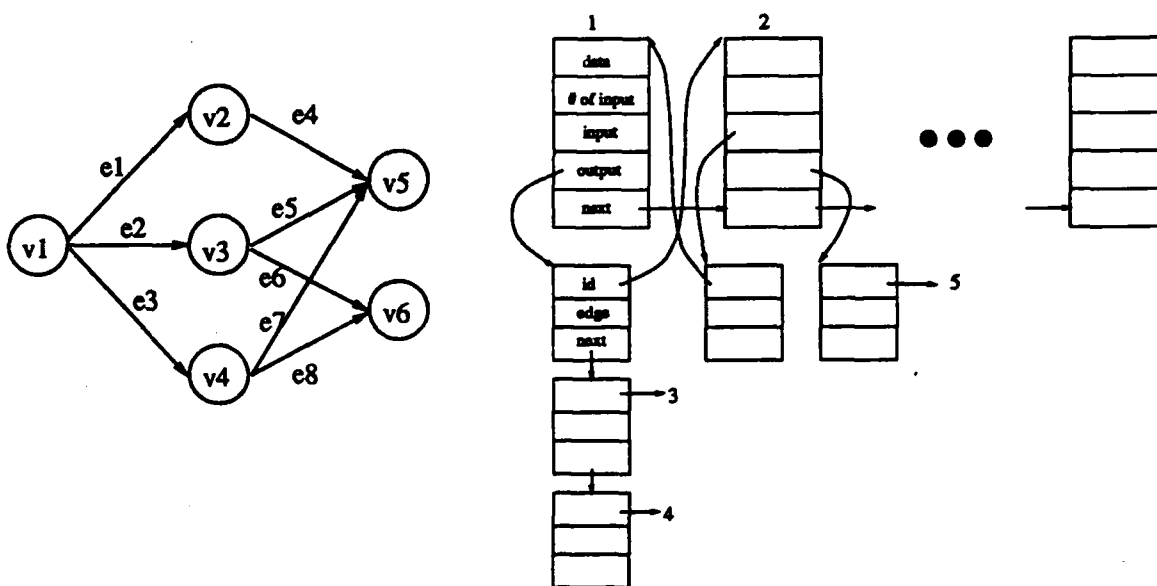


Figure 7.5: The data structure of a data dependency graph.

```
while (ifile) { // it becomes false at end of file
    ifile >> lexbuf;
    ...
}
```

To disconnect a stream attached to a file from the program, the translator invokes the function `close()`. For example,

```
ifile.close();
```

7.4.2 Internal Representation of Data Dependency

In this stage the translator detects a data dependency and gives a dependency between nodes by making a connection between `id` field of *InOut* structure and its corresponding *Node* structure. It then assigns variable type – integer or list – to local variable.

It visits all elements of the set of predecessors and successors of a node. In each element the translator tries to find the corresponding node and connects them. For example, if an element of the set of successors of node 1 forms the input of node 2, then the element of the set of successors of node 1 is linked to node 2. Similarly, the element of the set of predecessors of node 2 corresponding to the input from node 1 is linked to node 1. Figure 7.5 shows the data structure of a data dependency graph

and their connections. At that time the translator assigns the local variable to an element of the set of predecessors of the node.

After establishing the links, the translator identifies specific patterns representing *predicate* and *iteration* control structures. The nodes representing the *predicate* are replaced by a *predicate* node. The nodes representing the *iteration* are replaced by an *iteration* node. These replacements are accomplished following the schemes for detecting patterns as discussed in Section 7.1.2.

Once these replacements are done, the translator visits every node in the modified graph to fix the type of data flowing along each edge in the graph. In other words, the types of the local variables are fixed.

7.4.3 Code Generation of Method and Objects

In this stage the translator generates the target code for the data structure representing the classes, procedures representing methods, procedures representing objects, and the main procedure. It also generates the channel mechanism for communication between objects and the locking mechanism for synchronization.

The translator generates four kinds of files: `class.h`, `method.c`, `outXX.c`, and `main.c`. The file `class.h` is a header file which contains all the data structures required by the target code.

As mentioned before, the source language is PROOF/L and consists of objects encapsulating data and methods. All methods are identified from the IPR and generated in the file `method.c`. To generate the procedures representing methods from the IPR, the first step done by the translator is to identify both the input and output parameters of the procedure being generated. Once the parameters have been identified, the translator produces the code for variable declaration. The remaining step is to write the code for every method. This is achieved by following the translation rules mentioned in Section 7.1.

The files `outXX.c` consists of procedures representing **Passive**, **Pseudoactive**, and **Active** objects. Henceforth, we shall refer to such procedures as passive procedures, pseudoactive procedures, and active procedures, respectively.

Passive procedures act like service agencies. They wait passively until one of their methods is invoked by other procedures. This is achieved by using the Inmos C construct `ProcAlt()`. For example,

```
int signal;  
signal = ProcAlt(chan01, chan02, chan03, NULL);
```

`ProcAlt` suspends the current process until one of the channel arguments is ready to input. On completion, the functions return an index into the parameter list indi-

cating the ready channel. In the above example, it sets *signal* to 0, 1 or 2 according to which of the three channels becomes ready first. In Inmos C, channel variables represent unidirectional communication link between two processes. Channels between processes are created simply by declaring a variable type **Channel *** at an appropriate point in the program. Channel input and output functions are then used to pass data. Their functions must be paired for two processes to communicate and exchange data. Once the procedure sends data through a channel, it waits for a reply. In other words, these procedures are blocked until they are initiated by some other procedures.

For the sake of simplicity, we use a single-mode locking mechanism in implementation in order to ensure the consistency and correctness of objects. The locking mechanism supported by PROOF/L is as follows:

```
while (guard is false or lock is true) ; // busy waiting
lock = true;
sends data through a channel;
receives data through a channel;
unlock;
```

The guard associated with the method is evaluated. If the guard is False, repeat this evaluation until the guard is True. That is, the implementation of synchronization between objects uses "busy waiting" to achieve mutual exclusion. Then, it sets the lock and communicates data through a specified channel. Finally, it is unlocked.

Active procedures are active initially, and they may remain active throughout the execution except for occasional suspensions for the purpose of synchronization with other procedures. Each active procedure has its own body. The bodies of procedures are functions that may be recursive and diverse (non-terminating). They do not wait for an initiation by some other procedure to execute their body. These procedures can have any type of parameters with a restriction that the first parameter is a Process pointer. Channel variables for the communication between this procedure and other passive procedures are declared inside the procedure. An active procedure is capable of spawning a number of child processes and executing them in parallel. The body of an active object is translated by following the translation rules mentioned in the previous section.

The third and the final type of procedures are pseudoactive procedures. A pseudoactive procedure is a hybrid procedure consisting of an active part and a passive part.

In the file *main.c*, the translator generates the code for allocating the channels for inter-procedure communication. The code also allocates process pointers for the different types of objects mentioned above. Once these are done, the main process initiates the execution of these procedures in parallel.

7.4.4 Translation of Nodes

In this stage the translator generates the target code by translating each node in IPR as follows:

1. For a binary function node: "+", "-", "*", "/", "mod", it is translated by the simple function translation rule.
2. For a boolean function node: ">", ">=", "<", "<=", "EQ", "<>", "or", "and", "not", "True", "False", it is checked whether it comes from the predicate or iteration node. If true, it is translated into the condition statement of an "if" or a "while" statement. Otherwise, it is translated by the simple function translation rule.
3. For an identifier node which represents a user-defined function, it is translated to a procedure call by the simple function translation rule.
4. For an identity node, it is translated in two different ways depending on whether it is in the object or the method graph. If the identity node is in an object graph and has an input from another object, it is translated to the channel mechanism to receive data in the other object. After generating the channel mechanism, it is translated by the identity translation rule. Otherwise, it is translated directly by the identity translation rule.
5. For a constant node, it is translated by the constant node translation rule. That is, the translator generates the code assigning the constant value into the output local variable.
6. For a copy node, it is similar to an identity node. It is checked whether or not the input comes from another object. If so, it receives the input from the object. It is translated into the channel construct in Inmos C. Then, it is translated by the copy node translation rule. Otherwise, translation is achieved directly by the copy node translation rule.
7. For a construct node, the translator searches the suitable class which collects all the inputs into one class type, and it is translated such that each input element is assigned to a corresponding element of the composition in the class.
8. For a split node, it is opposite to a construct node. It is translated in such a way that each composition element of the input object is assigned to each local output variable.
9. For a latch, it is translated in such a way that the second input data is assigned to the output variable.
10. For an iteration node, it is translated to a "while" statement in Inmos C.
11. For a predicate node, it is translated to an "one-armed-if", "if-then-else", or "case" statement depending on the number of inputs.
12. For a macro node which represents a compound function composed of simple and/or macro functions, it is translated to a corresponding process in Inmos C.

Chapter 8

An Application Example

In this chapter, we give a hypothetical example to demonstrate our framework. The specification of the example is for the defense of a fictitious scenario of deployed air force bases.

8.1 Specifications for the Defense of Air Force Bases

Assume that there are three air force bases that are closely connected. For the sake of simplicity, we assume that only one type of fighters, one type of bombers, one type of surface to air missile batteries for defensive purposes against the attacking enemy. Radars and C3I (command, control, communication and intelligence) facilities are available. We assume that the equipment in a class has the same effectiveness. Effectiveness of the equipment is indicated on a scale of 1 to 100, and higher the effectiveness number, the higher the effectiveness of the equipment. Each base may have many radars, but has only one correlated radar value. Each base will also have several missile batteries and sufficient missiles to be used for its defense. Each base has either fighters only or bombers only, and hence a base is called either a fighter base or a bomber base, respectively. In our example, we have assumed that the bases are autonomous in their decision making process and thereby the functionality of the C3I had been embedded within the base. The distribution of the aircrafts in each base is given in Table 8.1. Table 8.2 gives the effectiveness values for both the friendly as well as hostile equipment.

It is assumed that the enemy aircrafts move at a speed of around 600 miles/hour. The radar has a range of about 1000 miles and gives the composition of the enemy cluster. Each enemy cluster is composed of either missiles only or a combination of fighters and bombers. It is assumed that at a given time the enemy sends no more than two clusters to attack a base. Furthermore, the enemy cluster is assumed to target a particular base and there is no sudden change in the course of any cluster by the enemy to attack a different base.

Table 8.1: Distribution of aircrafts in the base.

Base	Aircraft Type	Number of Aircrafts
Base 1	Bombers	20
Base 2	Fighters	40
Base 3	Fighters	50

Table 8.2: Effectiveness values for the friendly and hostile equipments.

Equipment	Friendly	Hostile
Bombers	60	60
Fighters	70	75
Missiles	85	80

If the enemy cluster consists of missiles only, then the bases defend themselves by launching their own missiles to intercept the incoming missiles. If the enemy cluster is composed of fighters and bombers, then the base defense strategy depends on the distance at which the enemy is detected. If the enemy is detected at a distance beyond 300 miles, then a fighter base tries to defend itself by launching its fighters and a bomber base defends itself by requesting help from neighboring fighter bases. If the bomber base does not get any help from its neighboring fighter bases, then the bomber base will defend itself with its missiles. For the sake of simplicity we have assumed that the bases are located such that each of them have only one neighboring fighter base. If the enemy is detected at a range between 50 and 300 miles, then the bases will defend themselves by using their missiles. If the enemy bombers are closer than 50 miles, the enemy bombers will release the bombs and it will be too late to defend the base. In such a case, the aircrafts in the base will fly out of the base and go to the aircraft shelters in a safer place, such as beyond the enemy's range. We assume that the minimum reaction time of the people protecting the base is about 40 seconds to get their aircrafts ready to flee. In addition, we assume that an aircraft can flee from the base at an average of one fighter every 5 seconds or one bomber every 10 seconds.

For a base to defend itself, it should calculate the effectiveness of the enemy equipment attacking it. The base then calculates the number of missiles or aircrafts needed to match the effectiveness of the enemy cluster and then sends the required number of aircrafts and missiles with at least the same total effectiveness. If the base is unable to match the enemy cluster with its own equipment, the base then requests help from its neighboring base. The neighboring base may or may not be able to help depending on how many aircrafts the neighboring base can spare. Only fighter bases can offer help.

8.2 Object-Oriented Analysis

8.2.1 Identifying Classes and Objects

We identify the following *Classes* from the requirement specification of the example:

- *Bomber_base* – for bomber base.
- *Fighter_base* – for fighter base.
- *Radar* – for radar.
- *Shelter* – a safe place that the aircrafts could escape, such as beyond the range of enemy aircrafts.

In addition, we add the following objects to the example to keep track of the operations in the bases:

- *Record* – to record the base operations
- *Reporter* – prints the data store *record*.

From the requirement specification we identify the following objects:

- *b1* – corresponding to one bomber base.
- *f2 and f3* – corresponding to the two fighter bases.
- *r1, r2 and r3* – corresponding to the radars for the three bases.
- *shelter* – corresponding to a safe place that the friendly aircrafts fly to when the enemy aircrafts get too close to the base.
- *record* – for recording base operations.
- *reporter* – for printing the data store.

8.2.2 Defining Class Interfaces

Class interface of an object consists of the input and output parameters and their types. Shown below are the class interfaces of the various objects identified in the previous subsection. As an example let us consider the class *Bomber_Base*. We show one interface which is called method *put_rad_value*. This method is invoked by the class *radar*. From the domain knowledge of the software we can infer that the radar value consists of the following details:

- Number of bombers attacking the base.

- Number of fighters attacking the base.
- Number of missiles homing in on the base.
- The distance of the enemy cluster from the base.

The type of the data is obviously to be integer and is the same as below. Thus, in a similar fashion the class interface for various classes can be determined. The classes Radar and Reporter do not have any methods that are accessible to others, and hence the class interfaces for Radar and Reporter do not exist. The complete class interface for a fighter base is as follows:

```
class Fighter_base
```

```
    method put_rad_value (f:Fighter_base, bomb:int, fight:int,
                        miss:int, dist:int -> Fighter_base)
    #called by the radar to pass the value of the enemy cluster
    #to the base.
```

```
    method help ( f:Fighter_base, n:int -> int )

    #invoked by a neighboring base when it needs fighters from
    #this base to defend itself.
```

```
    #to monitor the number of aircrafts on ground.
```

```
    method commit ( f:Fighter_base, n:int -> Fighter_base)
    method uncommit ( f:Fighter_base, n:int -> Fighter_base)
```

```
end class
```

8.2.3 Specifying Dependency and Communication Relationships Among Objects

Once the class interfaces are obtained for various classes, we can establish the dependency and communication relationship among the objects from the object-oriented analysis phase. Figure 8.1 gives the dependency and communication relationships among these objects. To illustrate the operation, let us consider the object *r1*. The object *r1* puts a radar value into the object *b1* and after doing so it records the radar values. Thus, there exists communication between *b1* and *r1* and between *r1* and *record*.

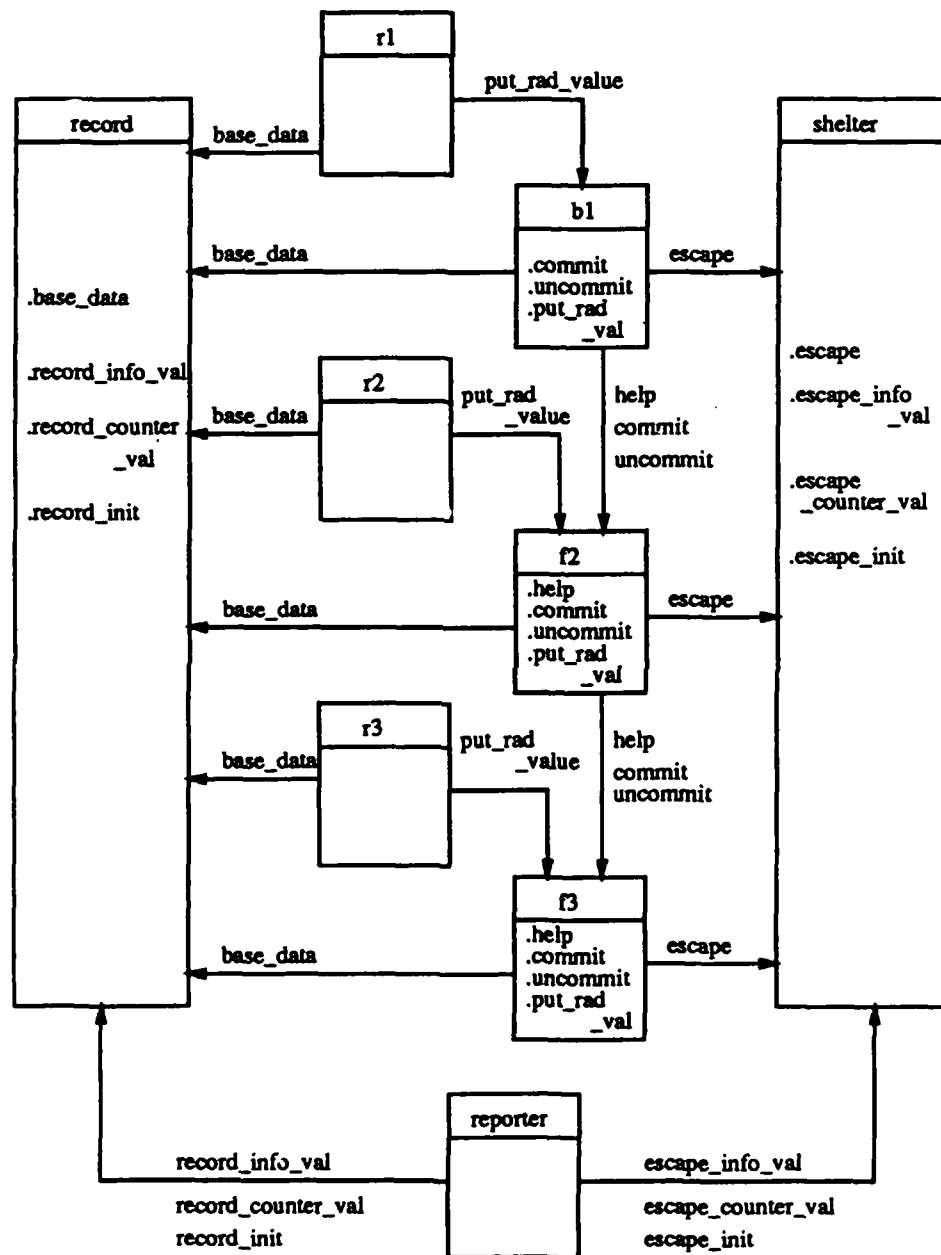


Figure 8.1: The object communication diagram for the set of decomposed objects of the example.

Table 8.3: Object Classification

Classification	Objects
Active	r1, r2, r3, report
Passive	record, shelter
Pseudo_active	b1, f2, f3

8.2.4 Identifying Active, Passive and Pseudo-Active Objects

From the requirement specification and from the object communication diagram shown in Figure 8.1, we can see that the object *r1* does not get invoked by other objects, but invokes *b1* and *record*. Thus, *r1* can be identified as an active object. To illustrate the method for identifying psuedoactive objects, let us consider the communication behavior of the object *b1*, which is invoked as well as invokes other objects. Hence, *b1* is identified as a psuedoactive object. If the communication behavior shows an object being invoked only, then it is identified as a passive object. A typical example is the object *Record*. We can classify the objects as shown in Table 8.3.

8.2.5 Identifying Shared Objects

By analysing the communication diagram as well as the object behavior for all the objects in this example, we can identify the objects *record* and *shelter* as shared writable objects. Shared objects are usually passive objects.

8.2.6 Specifying the Behavior of Each Object

From the notation used in Chapter 3, we can describe the behavior of each object. For instance, let us consider the object *r1*. As specified before, the object *r1* puts a radar value into the object *b1* and then records the radar value in the object *record*. The radar values as mentioned before are 1) the number of bombers attacking the base, 2) the number of fighters attacking the base, 3) the number of incoming missiles towards the base and 4) the distance of the hostile aircrafts or missiles from the base. These values can be generated concurrently. After these values are generated, *r1* records these values in the object *Record*, puts these values in the object *b1* and modifies the list of radar values it maintains. These operations can be done concurrently after the values are generated. Thus, we have the behavior of the object *r1*. The behavior of *r1* and *f2* objects are given below.

Behavior of object r1:

```
SEQ( CON(r1.generate_rad_bombervalue, r1.generate_rad_fightervalue,
```

```

        r1.generate_rad_missilevalue, r1.generate_rad_distancevalue),
    CON(b1.put_rad_value, record.base_data, r1.modify_list)
)

```

Behavior of object f2:

```

CON(
    #passive part of the base which waits for the help request from a
    #neighboring base.
    ONE-OF(WAIT(f2.help,f1),WAIT(f2.commit,f1),WAIT(f2.uncommit,f1)),

    #active part of the base which works on defending its own base.
    SEQ(WAIT(f2.put_rad_value, r2), f2.compute_range,
        SEL(
            #enemy too close. so escape
            CON(shelter.escape, record.base_data, f2.commit),
            #enemy in intermediate range. so use missile defense.
            record.base_data,
            #enemy far away. so check for enemy cluster.
            SEQ(f2.enemy_cluster,
                SEL(
                    #if missile attack use missile defense.
                    record.base_data,
                    #air craft attack
                    SEL(
                        #defend itself if possible
                        SEQ(f2.commit,record.base_data,f2.uncommit),
                        #ask for help from neighboring base
                        SEQ(f3.help,
                            SEL(
                                #if base can defend itself with
                                #its own aircrafts and with the
                                #neighbors help
                                SEQ(CON(f2.commit, f3.commit),
                                    CON(record.base_data,
                                        record.base_data),
                                    CON(f3.uncommit, f2.uncommit))),
                                #if no help available and base
                                #cannot defend itself with
                                #aircrafts, then use missile
                                #defense.
                                SEQ(f3.commit,
                                    CON(record.base_data,
                                        record.base_data),
                                    f3.uncommit)))))))))

```

Table 8.4: Object classification for the new set of objects

Classification	Objects
Active	r1, r2, r3, report
Passive	record1, record2, record3, s1, s2, s3
Pseudo_active	b1, f2, f3

8.2.7 Identifying Bottleneck Objects

We have already identified the objects *record* and *shelter* as shared writable objects. Since each of these objects are accessed by the three base objects b1, f2, f3, the access to the objects *record* and *shelter* have to be serialized. Hence, *record* and *shelter* are bottleneck objects. We can split each of them into three objects: *record1*, *record2*, *record3* and *s1*, *s2*, *s3*. In addition, *record1* and *s1* are associated with *b1*, *record2* and *s2* are associated with *b2*, and *record3* and *s3* are associated with *b3*. Thus, we have reduced the number of bottleneck objects in the system and enhanced the parallelism in the program.

Since each of the objects created in this step is only a copy of the existing objects, the class interface will not have to be modified. The object communication diagram will change to reflect the new objects and is shown in Figure 8.2. The new active, passive and pseudo active objects are given in Table 8.4. The shared objects are now only *record1*, *record2* and *record3* which are shared by the corresponding *radar* and base objects. This sharing is acceptable and will not create a bottleneck since the number of times a radar accesses the record object is small compared to the number of times the base object accesses the record object. The new object behavior *r1* and *f2* objects are as shown below.

Behavior of object r1:

```
SEQ( CON(r1.generate_rad_bombervalue, r1.generate_rad_fightervalue,
        r1.generate_rad_missilevalue, r1.generate_rad_distancevalue),
      CON(b1.put_rad_value, record1.base_data, r1.modify_list)
    )
```

Behavior of object f2:

```
CON(
  #passive part of the base which waits for the help request
  #from a neighboring base.
  ONE-OF( WAIT(f2.help,f1),WAIT(f2.commit,f1),WAIT(f2.uncommit,f1)),
```

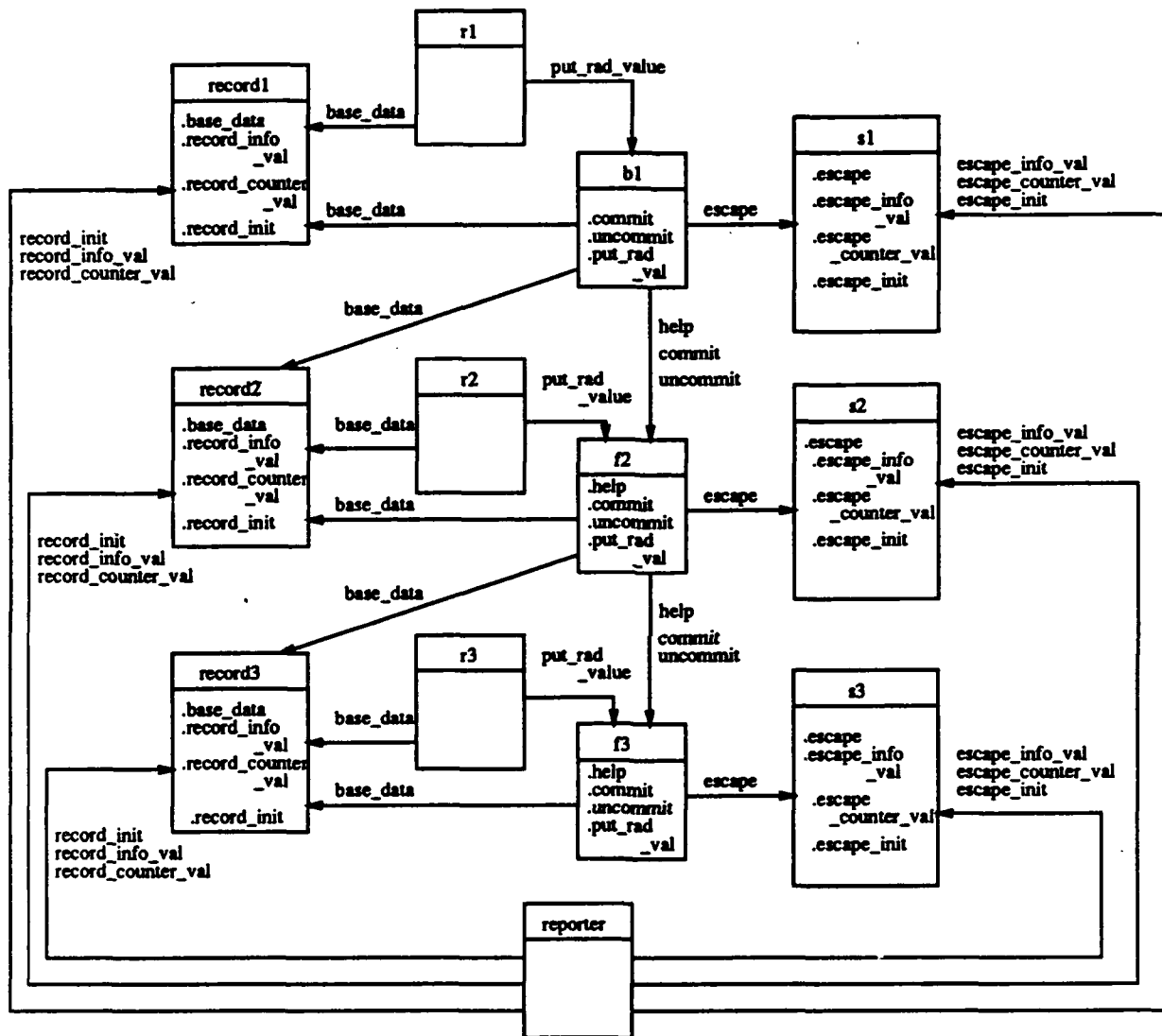



Figure 8.2: The object communication diagram for the modified set of objects.

#active part of the base which works on defending its own base.

```
SEQ(WAIT(f2.put_rad_value, r2), f2.compute_range,
    SEL(
        #enemy too close. so escape
        CON(s2.escape, record2.base_data, f2.commit),
        #enemy in intermediate range. so use missile defense.
        record2.base_data,
        #enemy far away. so check for enemy cluster.
        SEQ(f2.enemy_cluster,
            SEL(
                #if missile attack use missile defense.
                record2.base_data,
                #air craft attack
                SEL(
                    #defend itself if possible
                    SEQ(f2.commit, record2.base_data,
                        f2.uncommit),
                    #ask for help from neighboring base
                    SEQ(f3.help,
                        SEL(
                            #if the base can defend itself
                            #with its own aircrafts and with
                            #the neighbors help
                            SEQ(CON(f2.commit, f3.commit),
                                CON(record2.base_data,
                                    record3.base_data),
                                CON(f3.uncommit, f2.uncommit)),
                            #if no help available and base cannot
                            #defend itself with aircrafts, use
                            #missile defense.
                            SEQ(f3.commit,
                                CON(record2.base_data,
                                    record3.base_data),
                                f3.uncommit ))))))))
```

8.2.8 Checking for Completeness and Consistency of the Object-Oriented Analysis

By tracing through the behavior of the objects and also looking at the class interfaces we can easily see that the above object-oriented analysis is complete and consistent.

8.3 Object Design

8.3.1 Establishing Hierarchy

In this example, we have identified the fighter base and the bomber base as two different types of objects with some common behavior. Thus, we can define a super class called *Base* which has the information common to both the fighter base and the bomber base. We can thus make the *Fighter_base* and *Bomber_base* as derived classes from *Base*, inheriting all the information of the class *Base* in addition to their own special features. The remaining classes do not form a class hierarchy. Once the hierarchy is identified, the methods of each class are listed. The listing shown below describes the class heirarchy, various methods in a particular class and their functions.

```
class Base
```

```
    method put_rad_value (f:Base, bomb:int, fight:int,  
                        miss:int, dist:int -> Base)  
    #called by the radar to pass the value of the enemy cluster  
    #to the base.
```

```
    method compute_range ( f:Base -> int )  
  
    #returns a value proportional to the range.  such functions  
    #are used to for the sake of functional programming style.
```

```
    method enemy_cluster (f:Base -> int)  
  
    #determines if it is a missile or aircraft attack.
```

```
    method effective (f:Base -> int)  
  
    #effectiveness of the enemy cluster
```

```
    #to keep track of the aircrafts currently on ground.
```

```

        method commit ( f:Base, n:int -> Base)
        method uncommit ( f:Base, n:int -> Base)

end class

class Fighter_base : Base

    #the following methods return the number of aircrafts saved or
    #destroyed while escaping.

    method saved_values ( f:Fighter_base -> int)
    method destroyed_values ( f:Fighter_base -> int)



---



    method help ( f:Fighter_base, n:int -> int )

    #invoked by a neighboring base when it needs fighters from
    #this base to defend itself.

end class

class Bomber_base : Base

    #the following methods return the number of aircrafts saved or
    #destroyed while escaping.

    method saved_values ( b:Bomber_base -> int)
    method destroyed_values ( b:Bomber_base -> int)

end class

```

8.3.2 Designing Class Composition and Methods

The class composition typically consists of the local data present in the class. The type of the data present in the class is also identified. In this stage we also provide the methods present in each of the classes. As an example, consider the class composition of the class *Radar*. The data in the object are lists of predefined values which are integers for bomber values, fighter values, missile values and the distance of the enemy aircrafts and missiles. These constitute the class composition. In addition to these, we define the methods. The methods required for class *Radar* are 1) to generate the bomber value, 2) to generate the fighter value, 3) to generate the missile value, and 4) to generate the distance of the enemy aircrafts and missiles. These are defined formally below.

class Radar

composition

bomber:list(int) X	#predefined list of values
fighter:list(int) X	#for bombers, fighters, missiles
missile:list(int) X	#and the distance the enemy is
distance:list(int)	#currently detected

The following four methods are used to read the values
of the enemy cluster and distance from the predefined
list of values.

method generate_rad_bombvalue(r:Radar -> int)
expression
head r.bomber

method generate_rad_fightervalue(r:Radar -> int)
expression
head r.fighter

method generate_rad_missilevalue(r:Radar -> int)
expression
head r.missile

method generate_rad_distancevalue(r:Radar -> int)
expression
head r.distance

The following method is used to move the value read in
by the above methods from the Head of the list to the
tail of the list.

method modify_list(r:Radar, bomb:int, figt:int, miss:int,
dist:int -> Radar)
expression
delete the values from the head of their
corresponding list and append them to the
tail of that list.

end class

8.3.3 Designing the Body of the Objects

The body of an object describes the control thread within the body. A control thread exists for only active and pseudo-active objects. Thus, the body exists for only active and pseudo-active objects. In our application, the body thus exists for the objects *r1*, *r2*, *r3*, *b1*, *f2*, *f3* and *reporter* since these objects have been identified previously as active or pseudo-active objects. The behavior of the active objects should describe the body of that object. For example, the object *r1* has a body which iteratively executes in accordance with its behavior specified before and this is shown below.

Body of object r1:

```
SEQ( #Assign values to the radar object
    R[| r1 |] object radar( pre_assign radar values),

    while(True,
        #Generate the radar values to be passed on to the base.
        CON(r1.generate_rad_bombervalue, r1.generate_rad_fightervalue,
            r1.generate_rad_missilevalue, r1.generate_rad_distancevalue),

        #Put the values obtained above in the base and record, and then
        #modify the radar values. This operation will modify all the
        #objects involved.
        CON( R[| b1 |] b1.put_rad_value,
            R[| record1 |] record1.base_data,
            R[| r1 |] r1.modify_list)
    )
)
```

8.4 Verification

In the first step, the bodies of the active and pseudo-active objects are transformed into Petri nets. The transformation of the bodies of the objects in this application are shown in Figures 8.3 - 8.10.

The second step is to compose these nets to reduce the number of independent Petri nets. The nets are composed at the fusion point, also called the bottleneck place, so that shared modifiable objects are serialized for access among the different objects. For example, the object *record1* is a shared writable object that is modified by the objects *r1* and *b1*. Thus all the transitions in Figures 8.3 - 8.10 corresponding to the methods in *record1* are to be fused together at the bottleneck place. This process of fusing will bring the different nets together.

The last step is to refine the above nets to reflect the details of each method. This

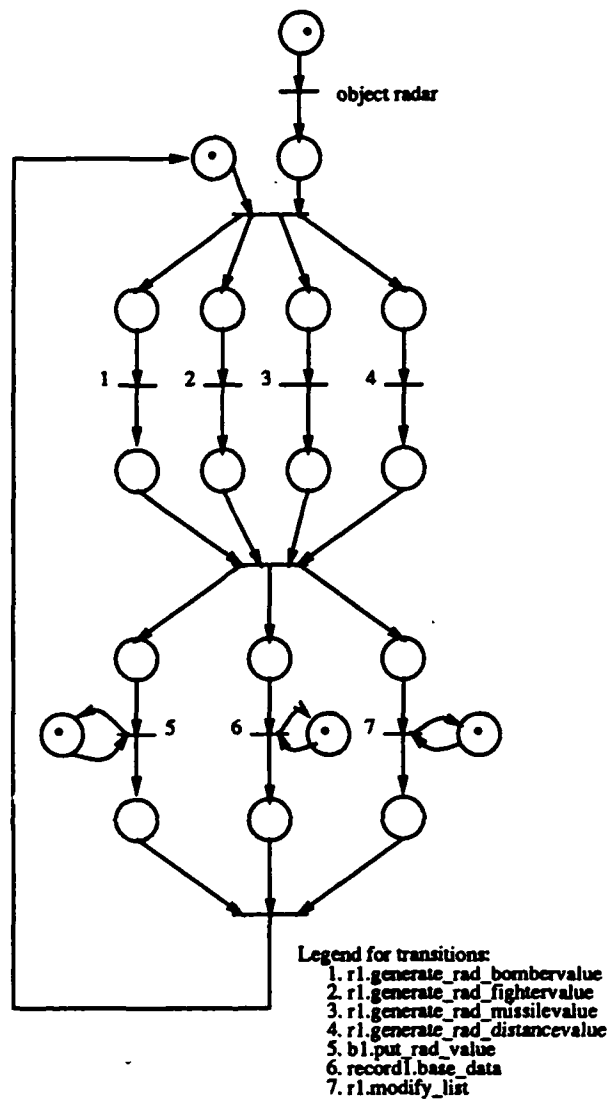


Figure 8.3: Transformation of *r1* to Petri net

is achieved by expanding each transition to show the guard and the expression. This has been illustrated earlier in our framework. Once the Petri net is obtained, we can then apply the available techniques to verify that the Petri nets satisfies the necessary properties.

8.5 Partitioning

The software system is decomposed as a set of the following objects: Radar *r1*, Radar *r2*, Radar *r3*, Shelter *s1*, Shelter *s2*, Shelter *s3*, Report reporter, Record *record1*, Record *record2*, Record *record3*, Bomber base *b1*, Fighter base *f1*, Fighter base *f2*. These objects are numbered from 1-13, respectively.

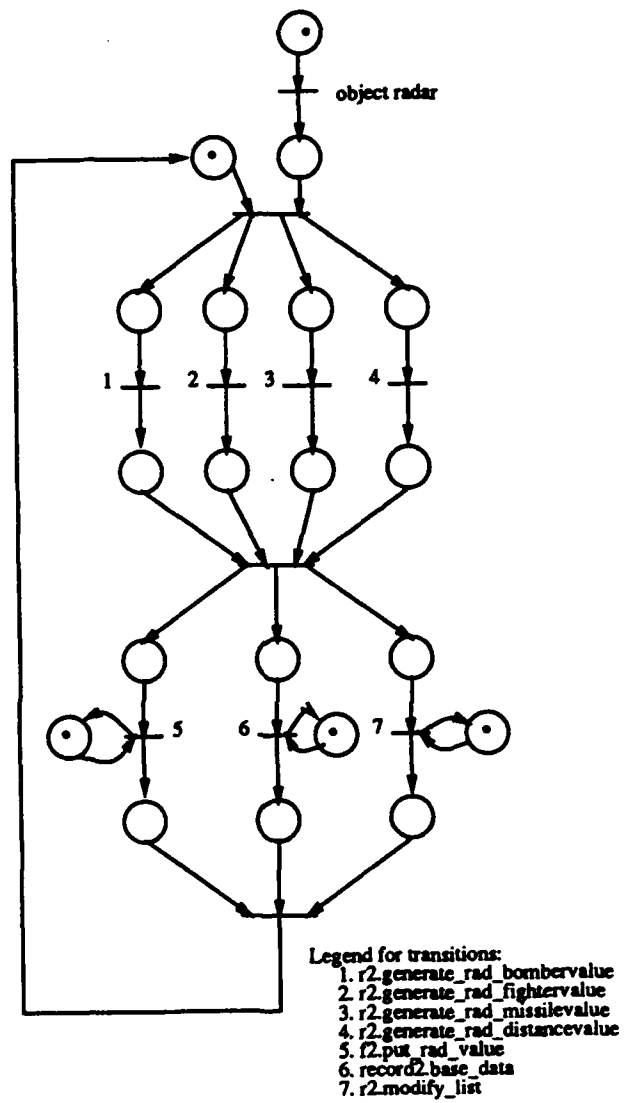


Figure 8.4: Transformation of *r2* to Petri net

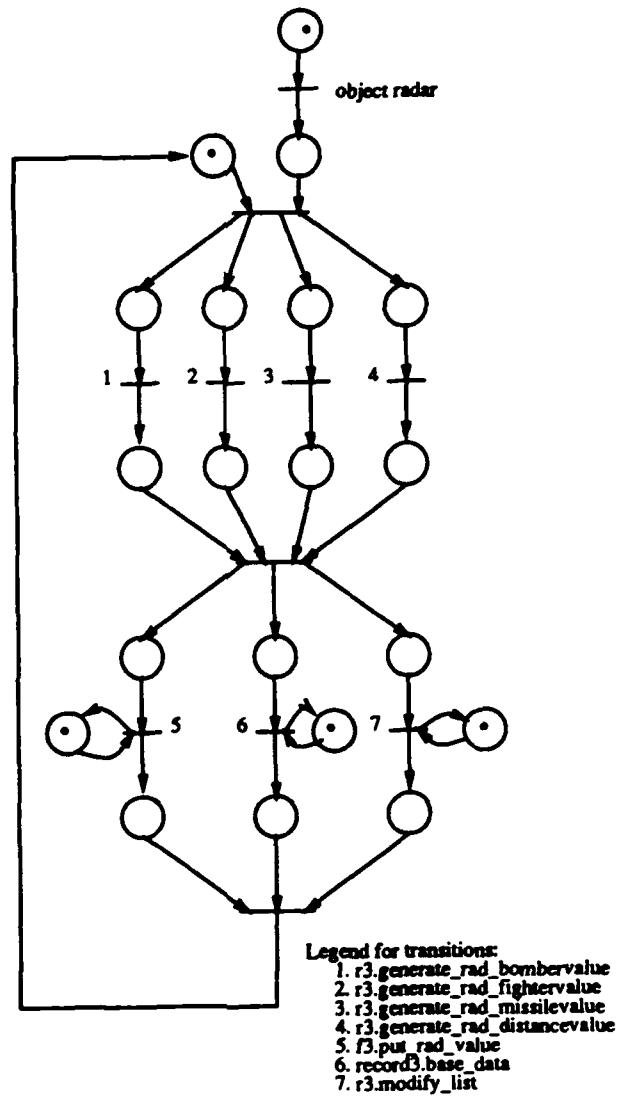


Figure 8.5: Transformation of *r3* to Petri net

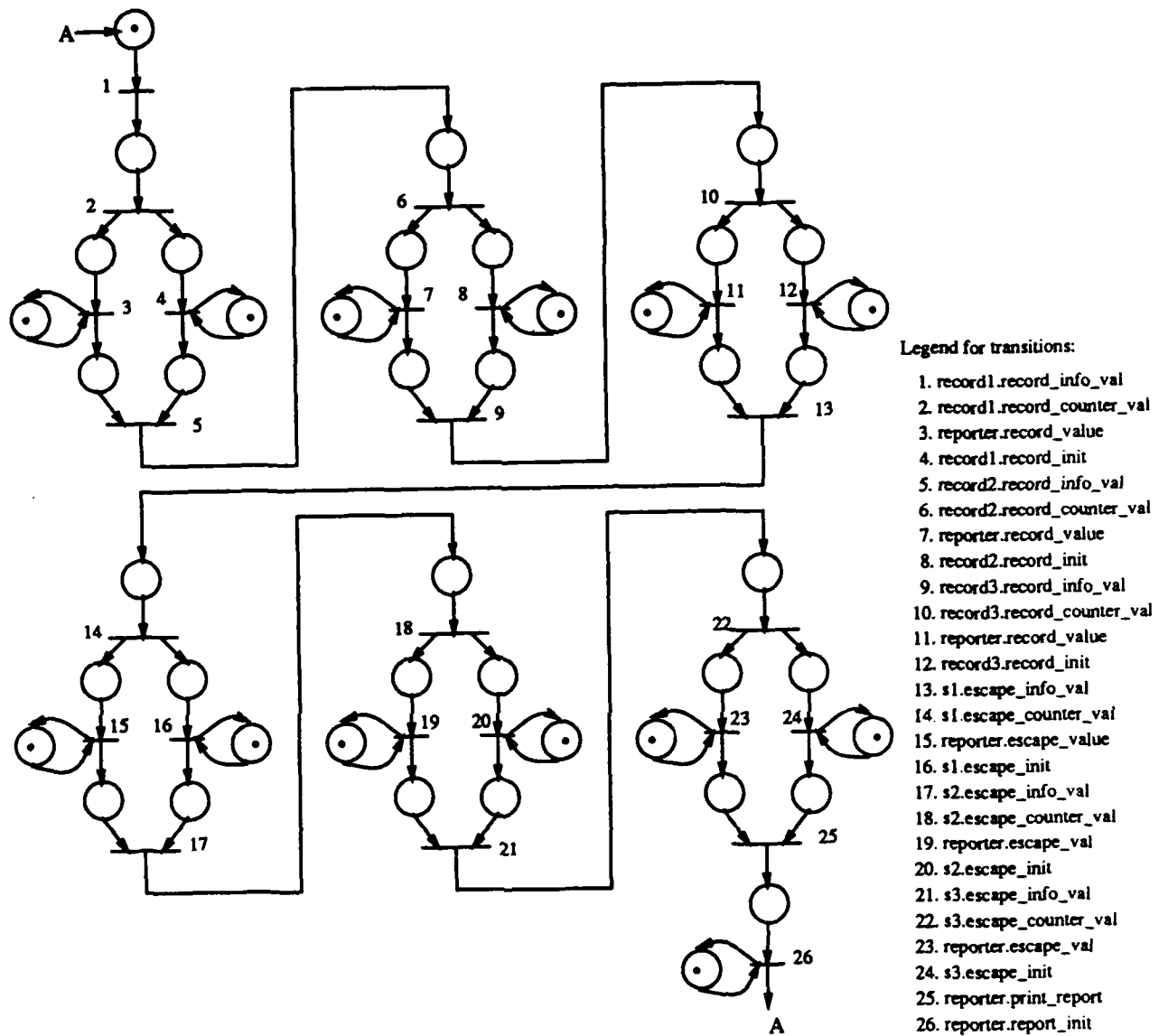


Figure 8.6: Transformation of *reporter* to Petri net

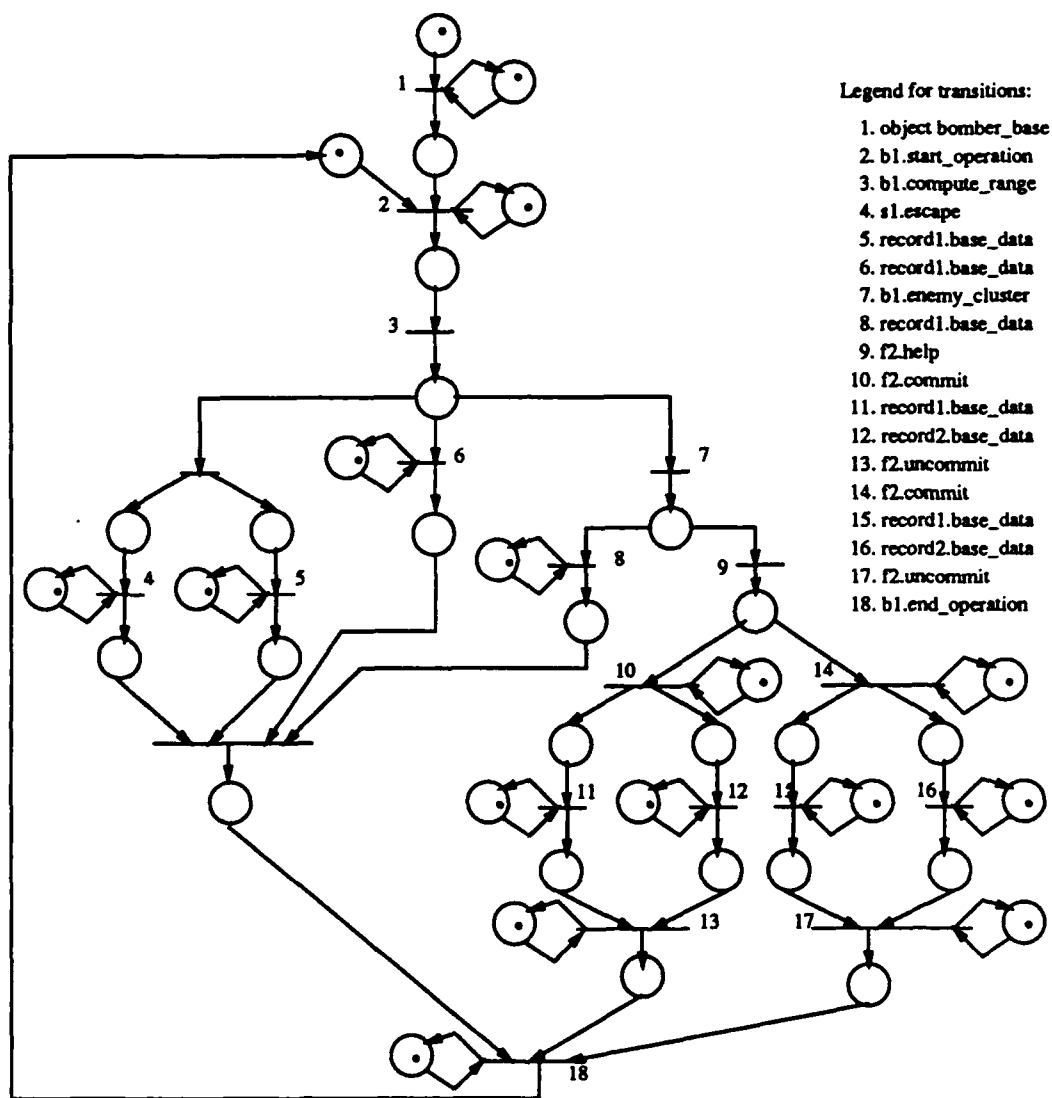


Figure 8.7: Transformation of *b1* to Petri net

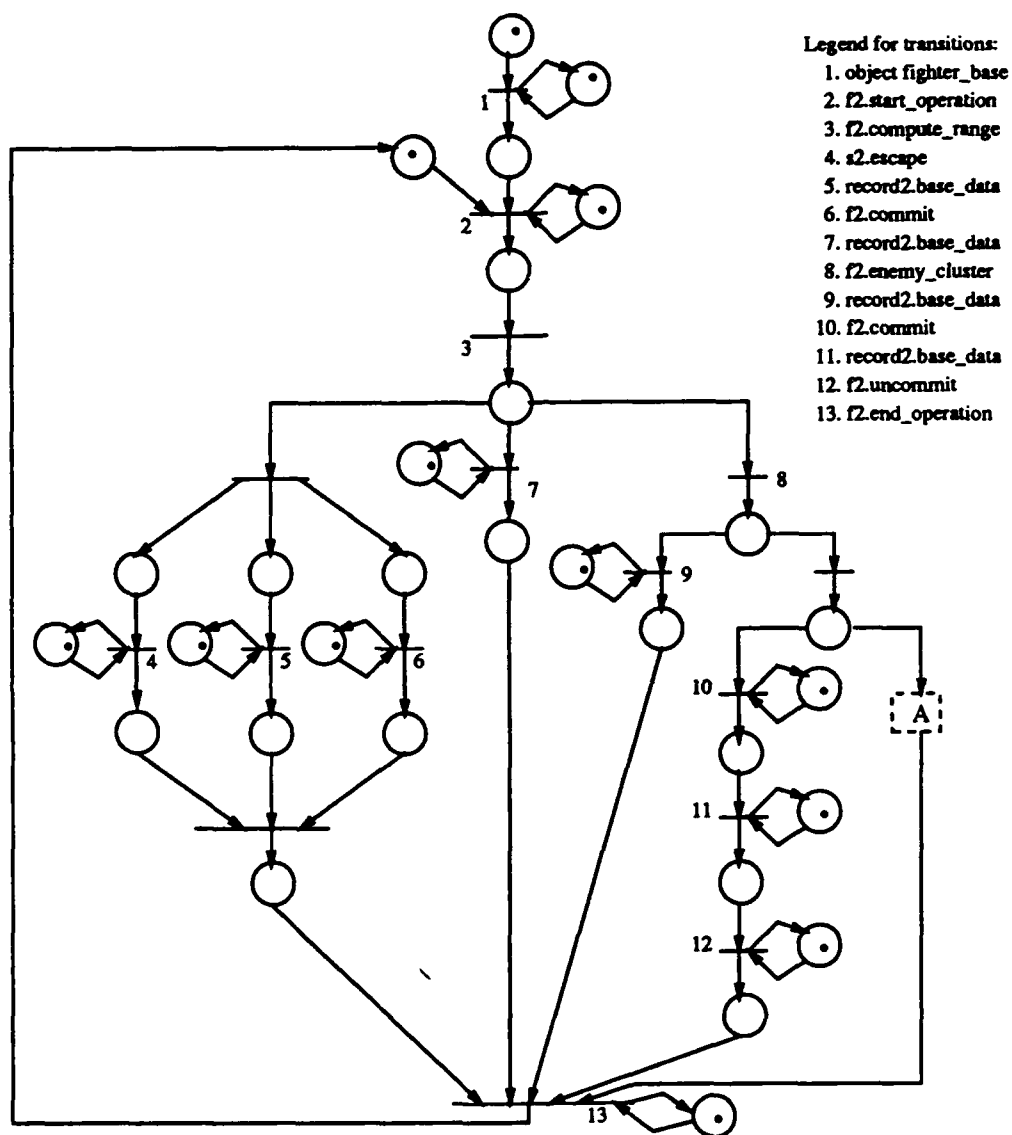


Figure 8.8: Transformation of $f2$ to Petri net

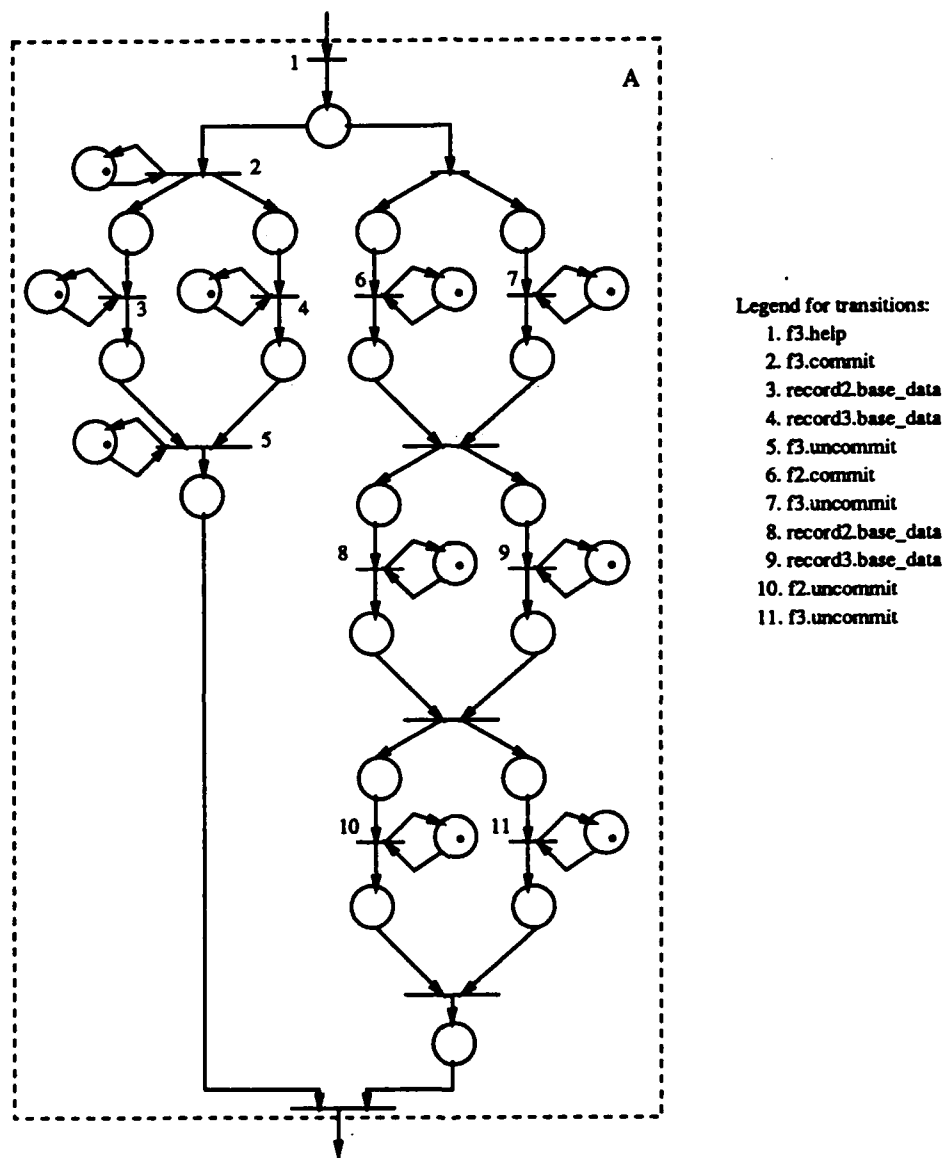


Figure 8.9: Transformation of *f2* to Petri net (cont.)

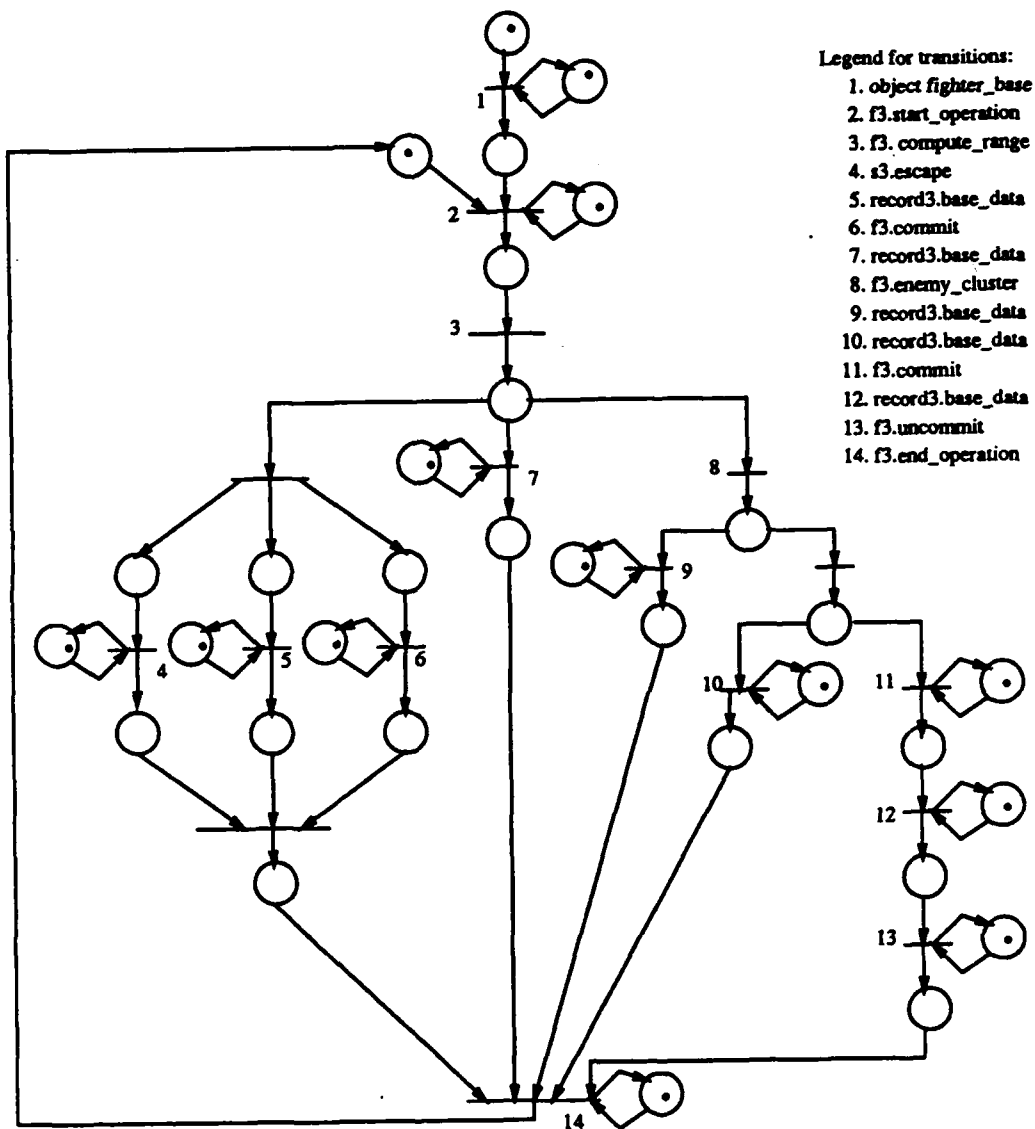


Figure 8.10: Transformation of *f3* to Petri net

The input includes the following three parts:

- Object behavior of the active and pseudo-active objects.
- The frequency of object invocations and the number of data units transferred between two objects every time one invokes the other.
- Number of replicated objects.

1) The object behavior is given as follows:

```
1  : SEQ( 1, CON( 11, 8, 1 ))
2  : SEQ( 2, CON( 12, 9, 2 ))
3  : SEQ( 3, CON( 13, 10, 3 ))
7  : SEQ( 8, 9, 10, 4, 5, 6 )

11 : SEQ( WAIT( 1 ),
        11,
        SEL( CON( 4, 8 ),
              8,
              SEQ( 11,
                    SEL( 8,
                          SEQ( 12,
                                SEL( SEQ( CON( 12, 8, 9 ),
                                              f1,
                                              CON( 12, 8, 9 ) ),
                                SEQ( CON( 12, 8, 9 ),
                                      f1,
                                      SEQ( 12, 8 )))))))))))

12 : CON( SEQ( WAIT( 2 ),
              12,
              SEL( CON( 5, 9, 12 ),
                    9,
                    SEQ( 12,
                          SEL( 9,
                                SEL( SEQ( CON( 9, 12 ),
                                              12,
                                              12),
                                SEQ( 13,
                                      SEL( SEQ( CON( 12, 13, 9, 10 ),
                                                  12,
                                                  CON( 13, 12, 9, 10 ) ),
                                      SEQ( CON( 13, 9, 10 ),
                                            12,
```

Table 8.5: Communication and Concurrency Weights for the Initial Graph.

-	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
(1)		-	-	-	-	-	-	(1,0)	-	-	(1,0)	-	-
(2)			-	-	-	-	-	-	(1,0)	-	-	(1,0)	-
(3)				-	-	-	-	-	-	(1,0)	-	-	(1,0)
(4)					-	-	(1,0)	(0,1/3)	-	-	(1/3,0)	-	-
(5)						-	(1,0)	-	(0,1/3)	-	(0,1/3)	(1/3,1/3)	-
(6)							(1,0)	-	-	(0,1/3)	-	(0,1/3)	(1/3,1/3)
(7)								(1,0)	(1,0)	-	-	-	-
(8)									(0,1/4)	-	(7/6,0)	(0,1/3)	-
(9)										(0,1/4)	(1/4,7/12)	(13/12,7/4)	(0,1/16)
(10)											(0,1/)	(1/13,12)	(1,37/24)
(11)											(1/2,25/12)	(0,1/6)	-
(12)													(1/4,2)

CON(13, 9))))))))

SEQ(WAIT(11), WAIT(11), WAIT(11))

13 : CON(SEQ(WAIT(3),
13,
SEL(CON(6, 10, 13),
10,
SEQ(13,
SEL(10,
SEL(SEQ(CON(10, 13),
13,
13),
10))))))
SEQ(WAIT(12), WAIT(12), WAIT(12))
)

2) When two objects communicate (i.e. one object invokes the other), the frequency of object invocation is assumed to be 10^3 and the number of data units transferred between the two objects every time is assumed to be 10^3 . Let $a = 10^6$ and $b = -10^3$ in the following discussion.

3) It is given that no replicated objects is included in the software system. As mentioned earlier, our partitioning approach can handle replicated objects used for satisfying fault tolerance requirements.

Figure 8.11 illustrates the initial graph and Table 8.5 contains the communication and concurrency weights for the initial graph.

Table 8.6 contains the edge weights at the end of the normalization stage.

In Table 8.6, the edge incident to nodes (12) and (13) has a minimum weight with

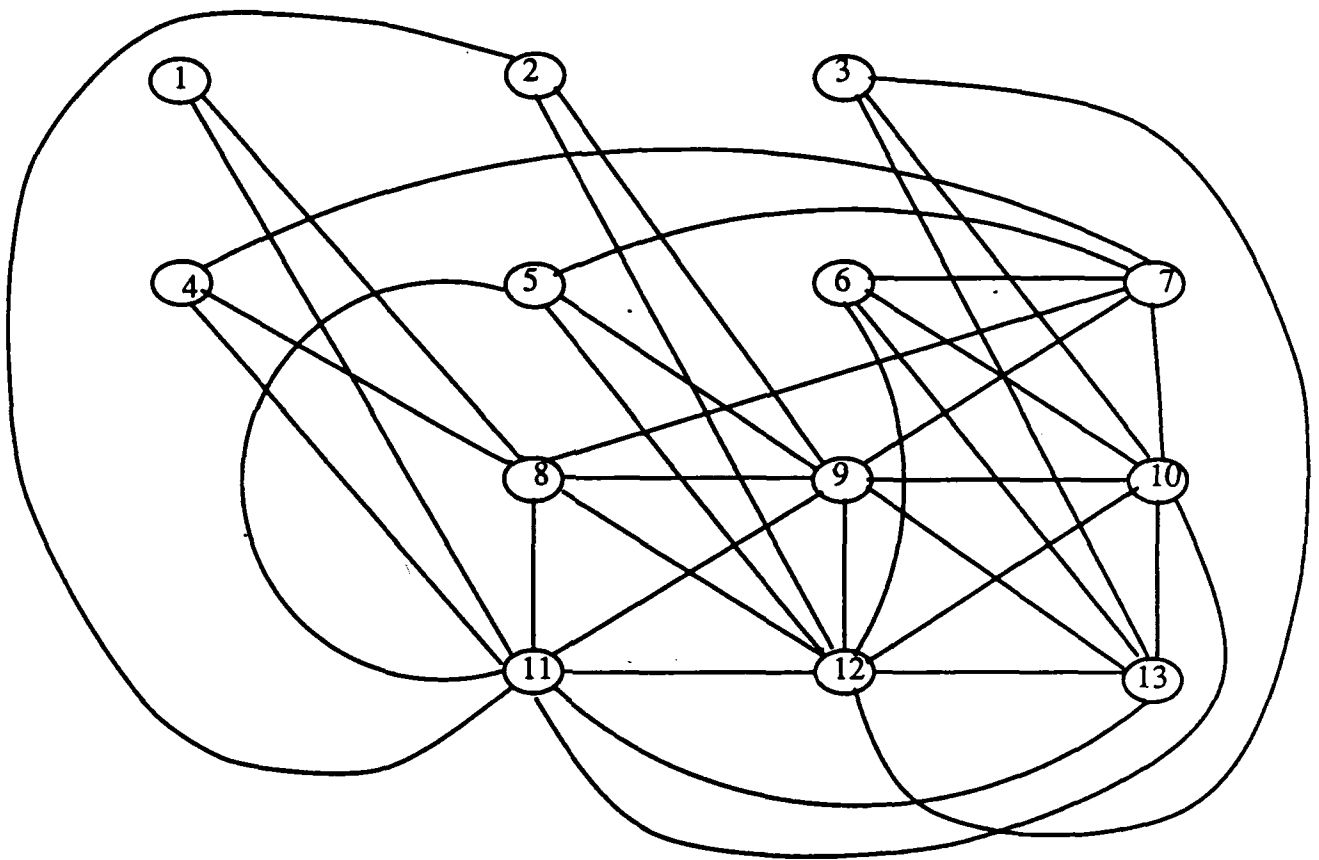


Figure 8.11: Initial Graph.

a negative value. Hence, the two nodes should be clustered. The resulting weights are tabulated in Table 8.7.

In Table 8.7, the edge incident to nodes (11) and (12, 13) has a minimum weight with a negative value. Hence, the two nodes should be clustered. The resulting weights are tabulated in Table 8.8.

In Table 8.8, the edge incident to nodes (10) and (11, 12, 13) has a minimum weight with a negative value. Hence, the two nodes should be clustered. The resulting weights are tabulated in Table 8.9.

In Table 8.9, the edge incident to nodes (9) and (10, 11, 12, 13) has a minimum weight with a negative value. Hence, the two nodes should be clustered. The resulting weights are tabulated in Table 8.10.

In the Table 8.10, the edge incident to nodes (6) and (9, 10, 11, 12, 13) has a minimum weight with a negative value. Hence, the two nodes should be clustered. The resulting weights are tabulated in Table 8.11.

In Table 8.11, the edge incident to nodes (5) and (6, 9, 10, 11, 12, 13) has a minimum weight with a negative value. Hence, the two nodes should be clustered.

Table 8.6: Weights after Normalization.

-	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
(1)		-	-	-	-	-	-	0.86	-	-	0.86	-	-
(2)			-	-	-	-	-	-	0.86	-	-	0.86	-
(3)				-	-	-	-	-	-	0.86	-	-0.48	0.86
(4)					-	-	0.86	-0.16	-	-	0.29	-	-
(5)						-	0.86	-	-0.16	-	-0.16	0.13	-
(6)							0.86	-	-	-0.16	-	-0.16	0.13
(7)								0.86	0.86	-	-	-	-
(8)									-0.12	-	1	-0.16	-
(9)										-0.12	-0.07	0.09	-0.08
(10)											-0.06	-0.41	0.26
(11)											-0.57	-0.08	
(12)													-0.75

Table 8.7: Gain Weights after the First Clustering.

-	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12, 13)
(1)		-	-	-	-	-	-	0.86	-	-	0.86	-
(2)			-	-	-	-	-	-	0.86	-	-	0.86
(3)				-	-	-	-	-	-	0.86	-	0.38
(4)					-	-	0.86	-0.16	-	-	0.29	-
(5)						-	0.86	-	-0.16	-	-0.16	0.13
(6)							0.86	-	-	-0.16	-	-0.038
(7)								0.86	0.86	-	-	-
(8)									-0.12	-	1	-0.16
(9)										-0.12	-0.07	0.01
(10)											-0.06	-0.15
(11)											-0.65	

Table 8.8: Gain Weights after the Second Clustering.

-	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11, 12, 13)
(1)		-	-	-	-	-	-	0.86	-	-	0.86
(2)			-	-	-	-	-	-	0.86	-	0.86
(3)				-	-	-	-	-	-	0.86	0.38
(4)					-	-	0.86	-0.16	-	-	0.29
(5)						-	0.86	-	-0.16	-	-0.03
(6)							0.86	-	-	-0.16	-0.03
(7)								0.86	0.86	-	-
(8)									-0.12	-	0.84
(9)										-0.12	-0.06
(10)											-0.21

Table 8.9: Gain Weights after the Third Clustering.

-	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10, 11, 12, 13)
(1)		-	-	-	-	-	-	0.86	-	0.86
(2)			-	-	-	-	-	-	0.86	0.86
(3)				-	-	-	-	-	-	1.24
(4)					-	-	0.86	-0.16	-	0.29
(5)						-	0.86	-	-0.16	-0.03
(6)							0.86	-	-	-0.19
(7)								0.86	0.86	-
(8)									-0.12	0.84
(9)										-0.18

Table 8.10: Gain Weights after the Fourth Clustering.

-	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9, 10, 11, 12, 13)
(1)		-	-	-	-	-	-	0.86	0.86
(2)			-	-	-	-	-	-	1.72
(3)				-	-	-	-	-	1.24
(4)					-	-	0.86	-0.16	0.29
(5)						-	0.86	-	-0.19
(6)							0.86	-	-0.19
(7)							0.86	0.86	-
(8)									0.72

Table 8.11: Weights after the Fifth Clustering.

-	(1)	(2)	(3)	(4)	(5)	(7)	(8)	(6, 9, 10, 11, 12, 13)
(1)		-	-	-	-	-	0.6	0.6
(2)			-	-	-	-	-	1.72
(3)				-	-	-	-	1.24
(4)					-	0.6	-0.16	0.29
(5)						0.6	-	-0.19
(7)							0.6	1.72
(8)								0.72

Table 8.12: Weights after the Sixth Clustering.

-	(1)	(2)	(3)	(4)	(7)	(8)	(5, 6, 9, 10, 11, 12, 13)
(1)		-	-	-	-	0.86	0.86
(2)			-	-	-	-	1.72
(3)				-	-	-	1.24
(4)					0.86	-0.16	0.29
(7)						0.86	2.5
(8)							0.72

Table 8.13: Weights in Output Graph.

-	(1)	(2)	(3)	(7)	(4, 8)	(5, 6, 9, 10, 11, 12, 13)
(1)		-	-	-	0.86	0.86
(2)			-	-	-	1.72
(3)				-	-	1.24
(7)					1.72	2.58
(4, 8)						1.01

The resulting weights are tabulated in Table 8.12.

In Table 8.12, the edge incident to nodes (4) and (8) has a minimum weight with a negative value. Hence, the two nodes should be clustered. The resulting weights are tabulated in Table 8.13.

In Table 8.13, no edge has a negative weight. Thus, Table 8.13 represents the weights in the output graph. The output graph is shown in Figure 8.12.

8.6 Implementation

8.6.1 Coding

Based on the design of the objects, we could write the code in the PROOF/L. After the first level of translation, we could perform the grain size analysis on the intermediate form generated from the first level of translation.

8.6.2 Grain Size Analysis

Since we are using the router in the application program, the communication time plays a major factor in determining the grain size. From the execution times of the various primitive nodes discussed before, we find that it is not feasible to exploit fine

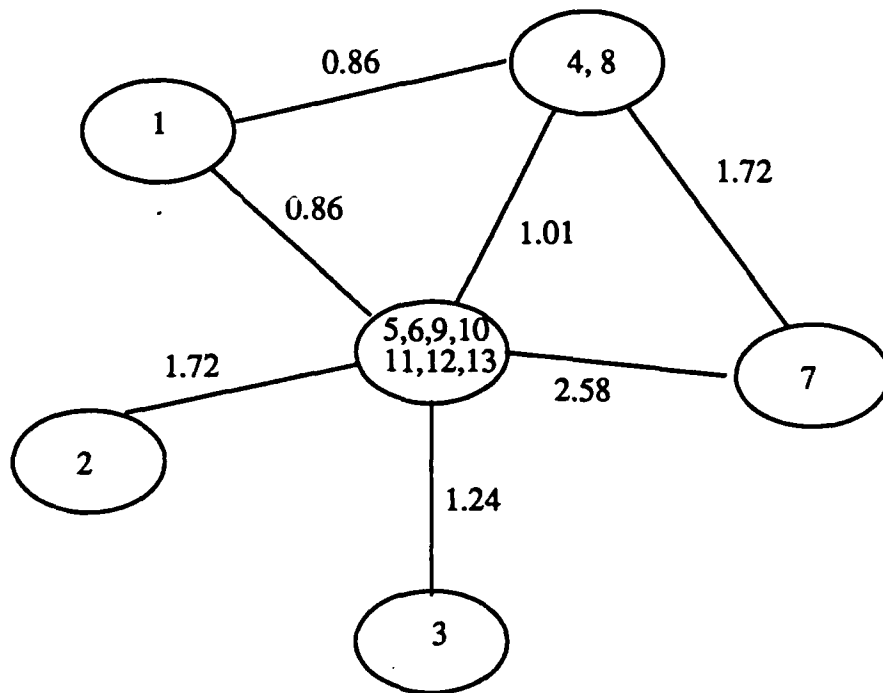


Figure 8.12: Output Graph.

grain parallelism on the transputer. This is because the communication time from one transputer to another is around 26 msec for a single byte of message, while the execution times of the primitive nodes in the intermediate form are far less. Thus, we have exploited parallelism at the object level to improve the performance of the overall system. The intermediate form for the application was modified such that it generated six files. In other words, there were six F-nodes in the modified intermediate form. Thus, the application was made to execute on six transputers.

8.6.3 Interfacing With the Router

The modified intermediate form generated above is then passed through the back-end translator and we get the INMOS C code. This target code is then interfaced with a router which removes the restriction on the feasibility of mapping Inmos C processes on the network of transputers because it allows the communication between processors which are not directly connected. To use the advantage brought about by the router, the output of the back-end translator requires to be interfaced with the router so that it can be executed.

Chapter 9

Conclusions and Future Research

In this project, an architecture transparent software development framework based on the parallel object-oriented functional computation model (PROOF) which incorporates the functional paradigm into the object-oriented paradigm has been developed. One advantage of the object-oriented paradigm is that it is a unifying paradigm where the concepts of the object-oriented paradigm are used from the analysis phase to the coding phase. Also, the object-oriented paradigm reveals the nature of the problem space naturally and this facilitates the mapping of any real world problems onto the parallel processing system. Our approach uses functional paradigm at the method level, which allows us to exploit massive parallelism. Thus, in our approach the coarse grain parallelism can be exploited at the object level while the fine grain parallelism can be exploited at the method level. Since we have separated the architecture dependent issues from the semantics of the program, the software system generated is much portable, thereby facilitating the implementation of the PROOF/L code on different target machines and target languages.

In order to ensure that our approach has distinct advantages over other methodologies in developing software for parallel processing systems, we will strive to evaluate the efficiency of our approach by comparing it with existing methodologies, such as dataflow oriented methodologies, based on criteria like code complexity, verifiability, portability, software development effort and availability of software tools. The impact on the efficiency of our approach by making it architecture independent will be also evaluated. Another direction of research we plan to do is to identify the class of architectures on which our approach will be very effective. Some of the classes to be studied are shared memory MIMD machines and distributed memory MIMD machines.

In order to make our approach useful, we will also develop PROOF/L to incorporate I/O facilities and data types. The I/O features include file I/O and standard I/O. File I/O includes reading, writing and appending to files. The data types include arrays, characters and floating point. We also plan to develop a translator for translating the IPR to NCC (NCube C), a language supported by the parallel processing system NCube. By doing this, we will not only take advantages of the available soft-

ware support for NCube, but also be able to evaluate the performance of the software generated by our approach on this machine. Furthermore, CASE tools need to be developed to aid the software developer in various stages in our approach, such as checking the consistency in the decomposition stage, design process and transformation of the body design into the corresponding Petri-Net models. The tools are also needed to aid the designer in the partitioning and the grain size analysis phases. We also need to develop a full-fledged compiler for PROOF/L.

The performance of the target code can be improved by developing optimization techniques. We have experimented our approach on various PROOF/L programs, including factorial problem, bounded buffer problem, dining philosopher problem, warehouse management problem and air-base defense simulation problem. We have compared the performance of the Inmos C codes generated by our transformation system with the performance of the Inmos C codes written by hand in terms of the completion time. Because the current transformation system does not include optimization techniques, the performance of the generated code is not as good as the code implemented directly on the transputer systems. For example, we have extensively tested pipelined versions of the factorial program. The speed-ups of the two factorial programs, the generated code and the directly written code by hand are almost the same, but the absolute completion time of the generated code is at best twice bigger than the completion time of the directly written code. When the programs involve extensive manipulation of list data types, the generated codes require much more time than the written codes. We believe that it is due to the inefficiency of current list handling routines implemented in the back-end transformation. In addition to the code optimization techniques, we need to develop techniques that can reduce unnecessary data movement during object invocation for improving the performance of the generated target code.

Bibliography

- [1] S. S. Yau, X. Jia, and D.-H. Bae, "PROOF: A Parallel Object-Oriented Functional Computation Model," *Jour. of Parallel and Distributed Computing*, Vol. 12, No.3, July 1991, pp. 202-212.
- [2] S. S. Yau, X. Jia, D.-H. Bae, M. Chidambaram, and G. Oh, "An Object-Oriented Approach to Software Development for Parallel Processing Systems," *Proc. 15th International Computer Software & Applications Conf., (COMPSAC 91)*, September 1991, pp. 453-458.
- [3] S. S. Yau, D.-H. Bae and M. Chidambaram, "A Framework for Software Development for Distributed Parallel Computing Systems," *Proc. Third Workshop on Future Trends of Distributed Computing Systems*, April 1992, pp. 240-246.
- [4] S. S. Yau, D.-H. Bae, and M. Chidambaram, "Object-Oriented Development of Architecture Transparent Software for Distributed Parallel Systems," To be published in *Computer Communications*.
- [5] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Computers*, C-21, No. 9, September 1972, pp. 948-960.
- [6] Ralph Duncan, "A Survey of Parallel Computer Architectures," *IEEE Computer*, Vol. 23, No. 2, February 1990, pp. 5-16.
- [7] Ardent Computer Co., *Programmer's Guide*, 1989.
- [8] Cray Research Inc., *Fortran (CFT) Reference Manual*, 1984.
- [9] C. Huson, T. Mache, J. Davies, M. Wolfe, and B. Leasure, "The KAP-205: An advanced source-to-source vectorizer for the Cyber 205 supercomputer," *Proc. of International Conf. on Parallel Processing*, 1986, pp. 827-832.
- [10] P. R. Fenner, "The Flex/32 for real-time multicomputer simulation," in W.J. Karplus, editor, *Multiprocessors and Array Processors*, Simulation Councils, Inc., 1987, pp. 127-136.
- [11] Arvind and K. Ekanadham, "Future scientific programming on parallel machines," *Jour. of Parallel and Distributed Computing*, Vol. 5, No. 5, Oct. 1988, pp. 460-493.
- [12] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," *Jour. of Parallel and Distributed Computing*, Vol. 5, No. 5, October 1988, pp. 617-640.

- [13] C.A.R. Hoare, "Communicating sequential processes," *Comm. ACM*, Vol. 21, No. 8, Aug. 1978, pp. 666-677.
- [14] P Brinch Hansen, "The programming language concurrent Pascal," *IEEE Trans. on Software Engineering*, Vol. 1, No. 2, June 1975, pp. 199-207.
- [15] Department of Defence, *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A-1983, 1983.
- [16] R. S. Nikhil, K. Pingali, and Arvind, "Id Nouveau," Technical Report Computation Structures Group Memo 265, Laboratory for Computer Science, MIT, 1986.
- [17] J. McGraw, "SISAL: streams and iteration in a single assignment language," language reference manual, version 1.2. Technical Report Technical Report M-146, LLNL, 1985.
- [18] K. L. Clark and S. Gregory, "PARLOG: Parallel programming in logic," *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 1, 1986, pp. 1-49.
- [19] H. Liberman, "Concurrent object-oriented programming in Act 1," In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, MIT Press, 1987, pp. 9-36.
- [20] S. S. Yau, X. Jia, and D.-H. Bae, "Trends in software design for distributed computing systems," *Proc. of the Second Workshop on the Future Trends of Distributed Computing Systems*, October 1990, pp. 154-160.
- [21] G. Booch, "Object-Oriented Development," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 2, Feb. 1986, pp. 211-221.
- [22] S. C. Bailin, "An Object-Oriented Requirements Specification Method," *Comm. ACM*, Vol. 32, No. 5, May 1989, pp. 608-623.
- [23] S. S. Yau, C.-C. Yang, and S. M. Shatz, "An Approach to Distributed Computing System Software Design," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 4, July 1981, pp. 427-436.
- [24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, NJ., 1991.
- [25] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.
- [26] S. S. Yau and M. U. Caglayan, "Distributed Software System Design Representation Using Modified Petri Nets," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 6, Nov. 1983, pp. 733-745.
- [27] S. S. Yau and C.-R Chou, "Control Flow Analysis of Distributed Computing System Software Using Structured Petri Net Model," *Proc. First Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, Sept. 1988, pp. 174-183.
- [28] F. Commoner, "Deadlock in Petri Nets," Wakefield, Applied Data Research, Inc., Report #CA-7206-2311, 1972.

- [29] P. Azema, G. Juanole, E. Sanchis and M. Montbernard, "Specification and verification of distributed systems using PROLOG interpreted Petri nets," *Proc. 7th International Conf. on Software Engineering*, Orlando, USA, 1984, pp. 510-518.
- [30] K. Lautenbach and H. A. Schmid, "Use of Petri nets for proving correctness of concurrent process systems," *Proc. IFIP Congress 74*, 1974, pp. 187-191.
- [31] H. Carstensen and R. Valk, "Infinite behavior and fairness in Petri nets," *Lecture Notes in Computer Science*, Vol. 188, 1985, pp. 83-100.
- [32] A. Datta and S. Ghosh, "Synthesis of a class of deadlock-free Petri nets," *Lecture Notes in Computer Science*, Vol. 31, July 1984, pp. 486-506.
- [33] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. IEEE*, Vol. 77, No. 4, April 1989, pp. 541-580.
- [34] F. Feldbrugge and K. Jensen, "Petri Net tool overview 1986," *Lecture Notes in Computer Science*, Vol. 255, 1987, pp. 20-61.
- [35] P. Coad, and E. Yourdon, "Object-Oriented Analysis", Yourdon Press 1990.
- [36] D. E. Eager, J. Zahorjan and E. D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Trans. on Computers*, Vol. 38, No. 3, 1989, pp. 408-423.
- [37] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, 1977, pp. 85-93.
- [38] C. C. Shen and W. T. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Computers*, Vol. 34, No. 3, 1985, pp. 197-203.
- [39] W. W. Chu, L. J. Holloway, M.-T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Vol. 13, No. 11, 1980, pp. 57-69.
- [40] O. I. El-Dessouki and W. H. Huan, "Distributed Enumeration on Network Computers," *IEEE Trans. on Computers*, Vol. C-29, No. 9, 1980, pp. 818-825.
- [41] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Vol. 15, No. 6, 1982, pp. 50-562.
- [42] P. R. Ma and E. Y. S. Lee, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Computers*, Vol. C-31, No. 1, 1982, pp. 41-472.
- [43] V. M. Lo, "Task Assignment to Minimize Completion Time," *IEEE 5th International Conf. on Distributed Operating Systems*, 1985, pp. 329-336.
- [44] S. M. Shatz, and S. S. Yau, "A Partitioning Algorithm for Distributed Software Systems Design," *Information Sciences*, Vol. 38, No. 2, 1986, pp. 165-180.
- [45] S. S. Yau and I. Wiharja, "An Approach to Module Distribution for the Design of Embedded Distributed Software Systems," *Information Sciences*, Vol. 56, 1991, pp. 1-22.

- [46] H. Kasahara and N. Seinosuke, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, Vol. C-33, No. 11, 1984, pp. 1023-1029.
- [47] S. J. Kim, "A general Approach to Multiprocessor Scheduling," University of Texas at Austin, No. GAX88-16494, 1988.
- [48] V. Sarkar and V. Hennessy, "Partitioning Parallel Programs for Macro-dataflow," *Proc. ACM Conf. in Lisp and Functional Programming*, 1986, pp. 202-211.
- [49] S. S. Yau, D.-H. Bae, and Gilda Pour, "A Partitioning Approach for Object-Oriented Software Development for Parallel Processing Systems," *Proc. 16th Annual Int'l Computer Software & Applications Conf. (COMPSAC92)*, October 1992, pp. 251-256.
- [50] M. R. Garey, and D. S. Johnson, *Computer and Interactability: A Guide to the Theory of NP-completeness*, Freeman, Sanfrancisco, 1979.
- [51] "The Transputer Databook," *Inmos databook series*, First edition, November 1988.
- [52] B. Goldberg, "Multiprocessor execution of functional program," *Jour. of Parallel Programming*, Vol. 17, No. 5, 1988, pp. 425-473.
- [53] C. McCreary and H. Gill, "Automatic determination of grain size for efficient parallel processing," *Comm. ACM*, Vol. 32, No. 9, 1989, pp. 1073-1078.
- [54] C. McCreary and H. Gill, "Efficient exploitation of concurrency using graph decomposition," *Proc. International Conf. on Parallel Processing*, 1990, pp. 199-203.
- [55] "An Intermediate Form Language IF1," Lawrence Livermore National Laboratory reference manual, 1985.
- [56] D. Pease, A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T. E. Karplinski, M. A. Mikki, and M. Zerrouki, "PAWS: A Performance Evaluation Tool for Parallel Computing Systems," *IEEE Computer*, Vol. 24, No. 1, January 1991, pp. 18-29.

**MISSION
OF
ROME LABORATORY**

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C³I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.